RWTH Aachen University

Master Thesis

Global Illumination methods in simulated VR environments

by

Valentin Belonogov

RWTH Aachen University

Master Thesis

Global Illumination methods in simulated VR environments

for the degree of M.Sc. in Media Informatics

by

Valentin Belonogov
Student Id.: 362 066

Prof. Dr. Torsten W. Kuhlen
Visual Computing Institute

Prof. Prof. Dr.-Ing. Jürgen Roßmann
Institute for Man-Machine Interaction

Supervisor: Alexander Atanasyan, M.Sc.

Date of issue: October 8, 2018

Statement

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig im Rahmen der an der RWTH Aachen üblichen Betreuung angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I guarantee herewith that this thesis has been done independently, with support of the Virtual Reality Group at the RWTH Aachen University, and that no other than the referenced sources were used.

Aachen, October 8, 2018 .....

Contents

CHAPTER 1

INTRODUCTION

1.1 Overview

Global illumination(GI) plays a considerable role in computer graphics. With the continuous improvement of hardware it has become attractive even for real-time applications nowadays. However, the calculation of physically plausible lighting is still expensive and faces challenges with performance in real time. Although it is worth noting the latest developments in solving this problem, for example such as Nvidia RTX [1]. On the other hand, the absolute correspondence to the physical model is not necessary to obtain a visually convincing result for many applications. Thus, in spite of the fact that many global illumination algorithms use precomputing and do not always work correctly with fully dynamic scenes, reasonable approximations can be used to achieve certain desirable results.

In this thesis, we decided to work with the algorithms based on ambient occlusion which allows to add realism to the image by calculating the intensity of light reaching a point of the surface. Unlike local methods, such as Phong shading, ambient occlusion is a global method, so the brightness value of each object point depends on the other objects in the scene.

These methods have different variations and provide a good compromise between image quality and performance.

1.2 Motivation and aims

The aim of this work was to study and implement a GI algorithm within the modular OpenGL-based renderer of an existing industry-grade 3D simulation system (VEROSIM) [2], which is provided for the work by the Institute for Man-Machine Interaction[1]. The framework provides us with a large model library as well as with interactive modelling tools for evaluation. It also supports generation of custom testing scenarios. The system

[1] https://www.mmi.rwth-aachen.de/

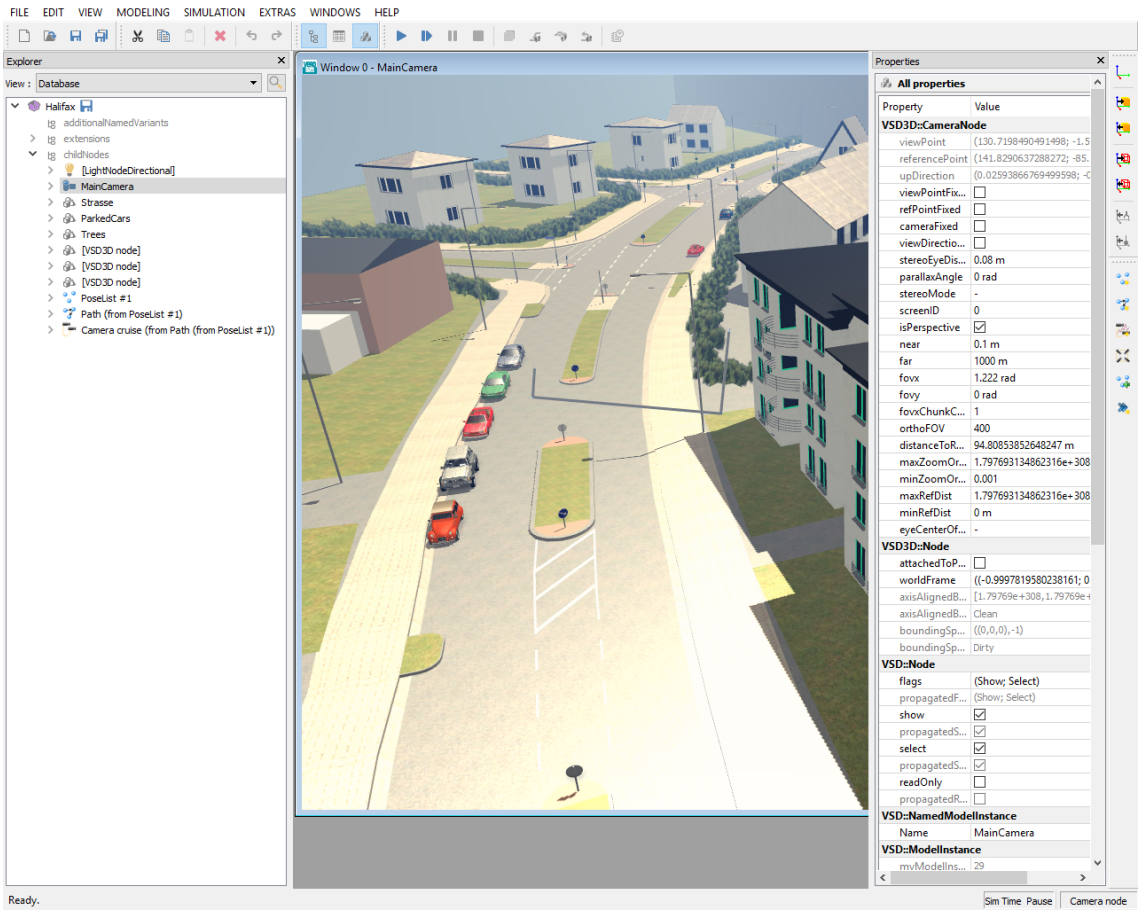

Figure 1.1: The VEROSIM GUI example. Here, the node tree structure of the loaded model is placed on the left side. The main window diplays a rendered view of the loaded scene. The menu on the hight-hand side displays the various properties of the currently selected node.

also features support for the major VR systems including support for openVR.

A focus of the work lies on the performance of the methods as real-time capability should be maintained. A qualitative as well as quantitative evaluation and comparison then is performed using appropriate metrics with respect to:

- visual appearance / resulting image quality
- performance
- usability for different simulation scenarios

1.3 Outline

This work is structured as follows.

In Chapter 2 the theoretical background needed for our work is introduced. We start with the description of the Rendering equation, continue with its approximation using direct and ambient lighting, then we use all the above to describe the general ambient occlusion approach.

Then, in Chapter 3, we describe the Screen Space Ambient Occlusion (SSAO) and Horizon-based Ambient occlusion (HBAO) with some of their variations, mention their advantages, disadvantages and ways to overcome them. The concrete implementation of these algorithms is detailed in the framework we use.

In Chapter 4, we discuss in detail the Screen Space Directional Occlusion (SSDO) technique, which includes two passes as direct light and indirect bounces. Its implementation and, importantly, the integration into the VEROSIM system is the main part of this thesis.

Furthermore, in Chapter 5, we present the performance measurements and comparative analysis of available algorithms. We carry out the evaluation with respect to different scenes, resolutions and variable parameters (eg sampling quality, radius and weight of occlusion). In particular, the efficiency of methods is evaluated in virtual reality conditions using the HTC Vive[2] head-mounted display.

Finally, in Chapter 6 we summarise the work we have done and briefly describe the directions of possible future research.

[2]https://www.vive.com/

CHAPTER 2

BACKGROUND

2.1 Rendering equation

In order to get a general idea of the ambient occlusion, it is necessary to understand the behavior of light and how objects reflect it.

In computer graphics, the interaction between objects and light is modelled by the so-called rendering equation which describes the propagation of light in a three-dimensional scene. In fact, it is an integral equation that determines the amount of light radiance in a certain direction as the sum of the intrinsic and reflected radiance. In other words, it is a description of how light reflects off a surface.

However, the rendering equation does not have an exact analytical solution. Therefore, it can be only solved by means of approximation. The degree to which the rendering equation is approximated determines how realistic the final rendering will be. Every computer graphics technique is trying to improve the solution of this problem and the spectrum of ambient occlusion algorithms is no exception.

The rendering equation can be obtained in an intuitive and straightforward way. As we can see in Figure 2.1, the viewer observes the point x on the given surface, and the goal of the rendering system is to calculate the amount of light reflected from x in the outgoing direction ω_0 to the viewer. The light reflected off x and towards the viewer is a function consisting of two components: the amount of light incident at x from every incoming direction ω_i and the properties of the surface which describing how the incoming light is reflected.

Surface properties are often described by the so-called "bidirectional reflectance distribution" function (BRDF). The BRDF can be understood as the answer to a question: How much light is reflected from a point in one direction due to the light coming in the other direction? As illustrated in Figure 2.2, this function has such parameters as the surface point x, an incoming light direction ω_i and outgoing light direction ω_r around a given point towards the camera, so it can be commonly denoted as $f(x, \omega_i, \omega_r)$. The return value is a fraction of reflected radiance exiting along ω_r due to the irradiance incident on the surface from direction ω_i.

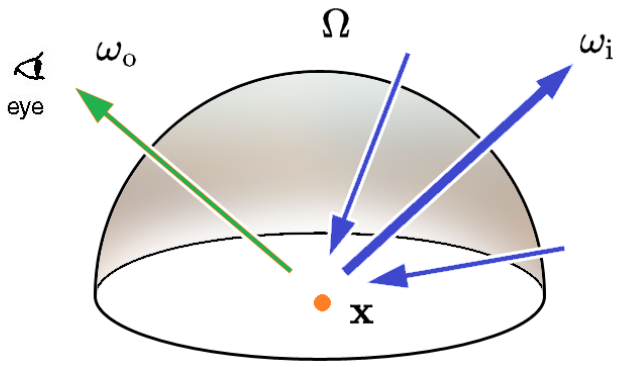

Figure 2.1: Light reflected at surface point x

Taking every beam of light that is incident at x in direction ω_i, multiplying this quantity by the BRDF and sum everything up we obtain the light reflected at x in direction ω_r. This process is exactly what is described by the rendering equation:

$$L_r(x, \omega_r) = \int_{\Omega} (f(x, \omega_r, \omega_i) L_i(x, \omega_i) cos\Theta_i d\omega_i \tag{2.1}$$

In the above equation, $L_r(x, \omega_r)$ denotes the light reflected at x in direction ω_r. This amount is the sum of the incident light $L_i(x, \omega_i)$ in direction ω_i multiplied by the BRDF $f(x, \omega_i, \omega_r)$ for every ω_i in the normal-oriented hemisphere, where ω_i is a differential solid angle in this hemisphere. The product is softened by $cos\Theta_i$ which is the cosine of the angle between incoming light ω_i and the surface normal. This cosine factor is needed due to Lambert's cosine law, which essentially states that light hitting the surface at normal incidence has a stronger influence on the reflected light than light coming at smaller angles, and this behavior is modeled by taking the cosine of the angle.

Looking closely at the equation described above, it becomes clear that the main difficulty lies in the computation of reflected light. L_r is a function of L_i as expressed by the equation, but at the same time L_i is also a function of L_r. So the part of the light reflecting off the surface bounces around the scene and finally falls back on the same surface.

Overcoming this complexity rendering environments approximate all these light bounces using different methods to achieve certain results. Methods that use more accurate approximations generally lead to higher quality of the visual appearance, but more expensive in terms of computation. So an important task of rendering systems is to find a balance between visual quality and overall performance.

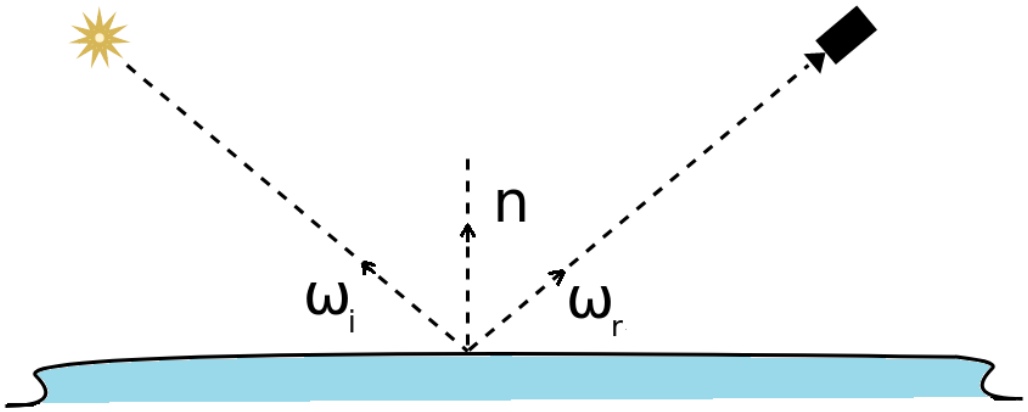

Figure 2.2: Vectors used to define BRDF. n is the surface normal,ω_i points toward the light source, ω_r points toward the camera. All three vectors are of unit length.

2.2 Conversion to Ambient Occlusion

The ambient occlusion can be considered as an approximation of the rendering equation. Next sections present a possible mathematical definition of the ambient occlusion. We start with simple steps adding more complexity further to achieve our goal.

2.2.1 Direct light

The simplest approach to simplifying the rendering equation is perhaps the complete ignorance of the indirect lighting and any light that bounces between surfaces as well. Only the light coming directly from the light source is taken into account during the computation. Using this agreement, as a result, we get images similar to Figure 2.3. As can be seen, only the points visible from the light source are lit, while the others remain completely dark.

This method is a rather coarse approximation of the rendering equation(2.1), and the visual appearence is not acceptable for most practical purposes. Practically we need to simulate some indirect lighting via bounces so the results would be visually plausible.

Increasing the number of bounces produces a better approximation of the rendering equation, but becomes computationally more expensive. Moreover, the computational costs are not proportional to the improvement final result quality. Even though modern rendering systems can support several bounces of indirect light [3] in real time they traditionally offer rendering equation approximations that are cheaper to compute and demonstrate qualitatively good results for the human eye perception.

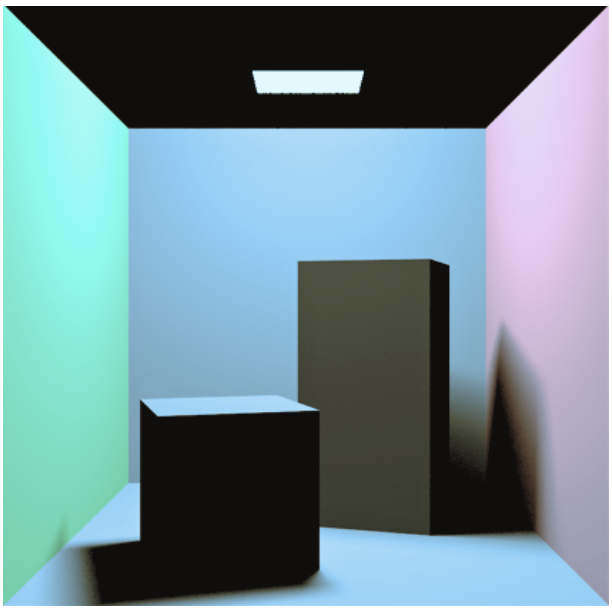

Figure 2.3: Cornell box model illuminated only with the direct light

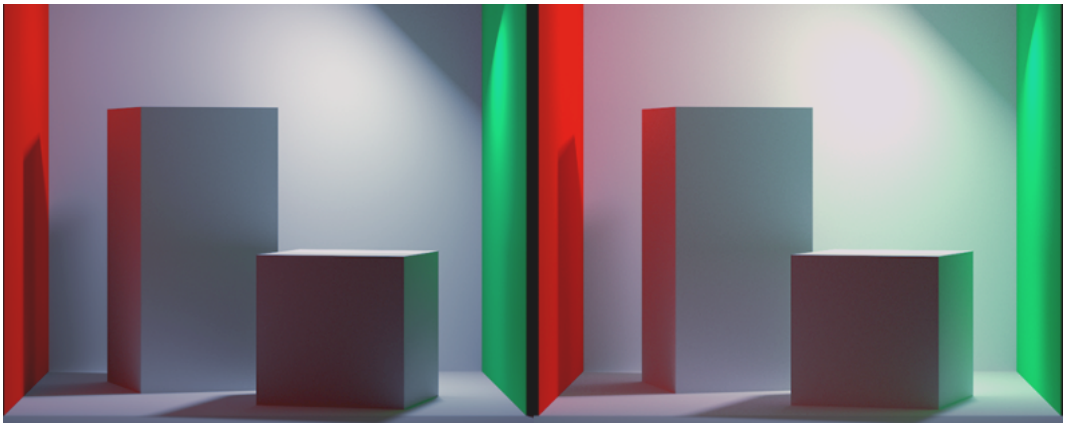

Figure 2.4: Cornell box model with 1(left) and 4(right) light bounces (rendered in KeyShot 3D Renderer)

2.2.2 Ambient light

The traditional computationally effective method of indirect illumination approximating is based on the replacement of all computation with a constant. Using a constant term we assume that light reaches all the points in every direction with the same intensity, ignoring the surrounding geometry objects. This term is called "ambient light" in some basic shading models, such as *Cook-Torrance* [4] or *modified Phong reflection model* [5].

Even though using the constant ambient term is cheap, it is not accurate and makes scene visually poor, and sometimes creates difficulties for the viewer to perceive the complexity of the shape of the objects in general. This is the exact problem that

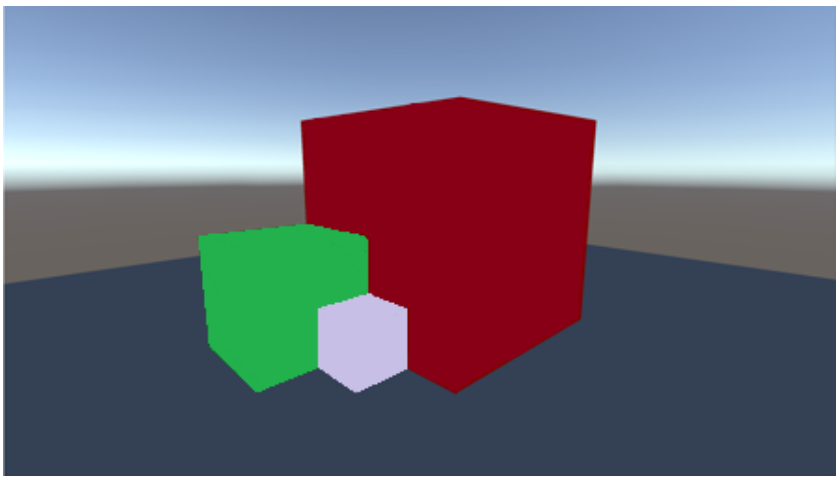

Figure 2.5: Ambient light illumination in Unity3D. Lighting only with the ambient component provides no depth cues.

ambient occlusion is trying to solve.

2.2.3 Ambient occlusion

We can treat the ambient occlusion as a form of indirect illumination that lies in-between ambient light and true indirect illumination, such as demonstrated in Figure 2.4. Instead of operating with a constant ambient factor, ambient occlusion generates an ambient value for each point in the scene. This value is represented by the function of the point visibility.

Turning to the ambient occlusion specifics it is important to note that it makes some assumptions when computing this ambient term.

One assumption implies that the surface we deal with reflects light identically in all directions. This is the so-called *perfectly diffuse* surface. Having this, we can move the surface BRDF out of the integral in redering equation because this component turns into constant now:

$$L_r(x, \omega_r) = b \int_\Omega L_i(x, \omega_i) cos\Theta_i d\omega_i, \tag{2.2}$$

where b stands for $f(x, \omega_r, \omega_i)$ BRDF function.

The second assumption states that if the light hits a point x then it happens equally in all directions and does not participate in any other more expensive illumination effects such as caustics or inner reflections. So the intensity of transported light is equal for every direction, and the direction itself actually becomes irrelevant in this case. In such way, ambient occlusion is essentially non-directional. Later, in Chapter 4, we will drop this constraint using the method of directional occlusion which will open up new possibilities for improvements.

Since ambient occlusion takes into account the point's visibility, for every direction ω_i in which light may theoretically reach given point x, ambient occlusion determines whether a ray in direction ω_i hits some other surface. If this is the case then point x is considered to be occluded in that particular direction and the light contribution is not added. Otherwise, if the ray does not meet any obstacles on the way to the surface, we take this into account and operate with it in the final calculation of the reflected light L_r.

Now we can express it using the equation:

$$L_r(x, \omega_r) = b \int_\Omega Visibility(x, \omega_i) cos\Theta_i d\omega_i, \tag{2.3}$$

The visibility function $Visibility(x, \omega_i)$ takes boolean values depending on the presence of shading: 0 if the point x is occluded in direction ω_i and 1 otherwise.

Now let's find the value of the constant b so that the final ambient occlusion value would lie in the range from 0 to 1. Rewriting the equation with respect to only non-occluded point we have:

$$L_r(x, \omega_r) = b \int_\Omega cos\Theta_i d\omega_i = b\pi \tag{2.4}$$

Since it should be equal to 1 for non-occluded case then the term b equals to $\frac{1}{\pi}$ and eventual AO equation becomes:

$$L_r(x, \omega_r) = \frac{1}{\pi} \int_\Omega Visibility(x, \omega_i) cos\Theta_i d\omega_i \tag{2.5}$$

This ambient occlusion definition however causes another problem. It applies to cases when we are dealing with a closed scene. For such scenes the ambient occlusion value is equal to 0 because the rays from any point x always hit some other surface. Thus the visibility function is nullified for every direction ω_i.

To overcome this error, the $Visibility(x, \omega_i)$ factor has to be substituted by a distance function $d(x, \omega_i)$. This function calculates the distance from point x to the potential occluder in direction ω_i. Then the this function goes through the distance test t to be classified as an occluder.

$$L_r(x, \omega_r) = \frac{1}{\pi} \int_\Omega t(d(x, \omega_i)) cos\Theta_i d\omega_i \tag{2.6}$$

The above equation describes the distant limited variation of standard ambient occlusion. The distance test $t(d(x, \omega_i))$ has some predefined threshold to detect the occluders:

$$t(d(x,\omega_i)) = \begin{cases} f(d(x,\omega_i)) \in [0,1] & \text{if } d < \text{threshold} \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

Sometimes this technique is called *ambient obscurance* [6] and has following features. Method can be applied to closed scenes and it returns 0 only for occluders that located further than some threshold. Also it deals with relatively local computation because $t(d(x,\omega_i))$ drops far away points and considers only nearby geometry. This contributes to more efficient solutions in practice. Finally, function $t(d(x,\omega_i))$ sets a greater occlusion value for the nearest occluders than for the distant ones. It is also worth noting that this variation in general has a similar essence of what was described before. Thus, this term is mentioned in literature more often as just *ambient occlusion*.

To obtain numerical results, ambient occlusion method can be implemented by casting rays to all directions within the hemisphere using the Monte Carlo random sampling method. However, ray-casting still requires much computational power, so an alternative approaches were introduced. In order to achieve higher frame rates these methods compute ambient occlusion less accurately. Instead of processing all the surfaces in 3D, they frequently approximate ambient occlusion in the image-space.

CHAPTER 3

AMBIENT OCCLUSION APPROACHES IN VEROSIM

3.1 VEROSIM Rendering Extenstion

Turning to VEROSIM, the implementation is presented as a rendering extension plugin modification. The new class is derived from `VSLibRenderGL::RenderExtension`. The extension is called in the rendering process execution and can thus render content to the screen. Derived class augments the functionality of base class by adding the properties and methods to suit the necessary requirements. The base class, in turn, provides all related information, GL functions and variables. The rendering extension is also assigned to the specific layer where it is going to be rendered.

VEROSIM framework supports The OpenGL Extension Wrangler Library, so the rendering plugin supports OpenGL 4 vesion. However, "Compatibility Profile" provides the backward compatibility preservation.

The VSPluginRenderGLAmbientOcclusion plugin was taken as the basis, which was supplemented in order to expand the rendering techniques it provides.

3.2 Screen Space Ambient Occlusion

3.2.1 Description

Screen-space ambient occlusion is an approximation of ambient occlusion that is carried out in the screen space in order to increase the performance and achieve real-time framerate [14]. The main idea of this method is to use the depth buffer to obtain a coarse representation of the scene geometry. The buffer values are already known, because they have been computed during first rendering stage. So the algorithm can be computed in a post-processing step.

The algorithm samples an area around every pixel. Then these samples are used to provide the ambient occlusion estimation of a pixel. This estimation determines the

visibility factor of the point that is based on the differences between the sample and point depths. Same as in ambient occlusion, this factor reduces the light strength in corners and other hollow parts of a scene. It can also be applied to diffuse and specular components, but experiments have shown that the best result is achieved with respect to the ambient one [9]. As seen in Figure 3.1, the more samples fall inside geometry, the less ambient lighting the particular fragment receives.

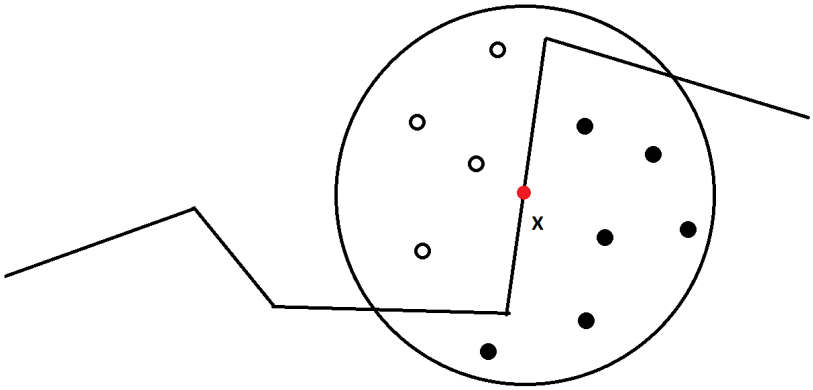

Figure 3.1: Sampling points around point X. Black samples that are inside the surface determine the final occlusion rate.

If the area around the pixel is sampled randomly, some parts of the scene(for example walls adjusted along the view direction). Information about the normals can help to solve this problem. Using additional buffer values allows to generate normal-oriented hemispheres for all pixels (Figure 3.2). This approach is used in the implementation in the VEROSIM.

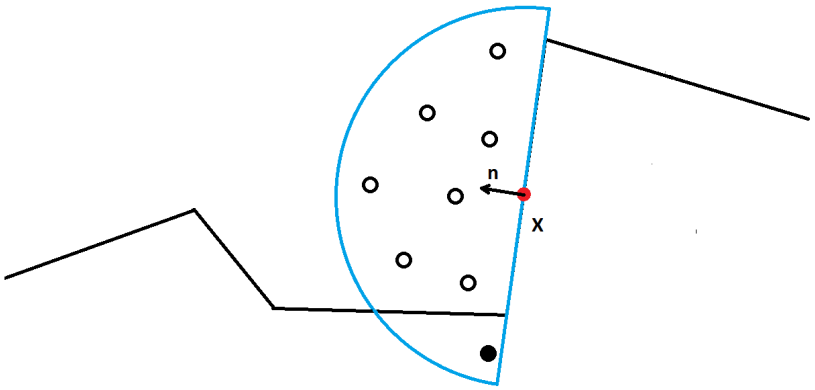

Figure 3.2: Normal-oriented hemisphere around point X.

The quality and accuracy of the final result directly depends on the number of

samples in a kernel. In the case of a few number of samples per pixel we lose accuracy and *banding* effect pops up. It generates visible stripes or bands in the image that are visually disturbing. By increasing the size of the kernel, the precision of calculations rises accordingly. However, this is inversely proportional to performance. To keep the quality with a relatively small number of samples, we can rotate the sample kernel by some random value. This randomness brings a noticeable noise, which is subsequently corrected by blurring the texture(a detailed description of smoothing techniques is provided in chapter 4).

It is also important to note that SSAO eventually deals not with the three-dimensional hemisphere but with its disc-shaped projection on the image space. This approximation of the hemisphere around the point is sensitive to the initially defined parameters, such as depth difference where occluders are considered or the sampling area.

3.2.2 Limitations

One of the artifact that arises from the screen-space limited approaches is *popping* [10]. Since the method works in screen space, objects that are not in view of the the camera do not create any occlusion on the nearby geometry. However, when the camera moves and reveals those objects, the occlusion is applied to other surfaces. As a result, new shadows appear in the image. Likewise, when geometry diappears from the scope, the shadowing factor vanishes.

Another artifact caused by screen-space methods is *flickering*. Since SSAO samples a number neighboring of pixels around the original point, when the camera moves, the sampled pixels belong to different three-dimensional point positions in the scene. So the occlusion can get slightly different factor with respect to the same point. As a result, pixels may flicker during animation.

Thus, it can be said that SSAO is view dependent while general definition of ambient occlusion does not have such limitation. To reduse flickering and popping artifacts, Guard-Band Clipping technique can be used [11]. It is based on an increase of the off-screen viewport size partially allowing processing of the objects that are not displayed (Figure 3.3).

3.2.3 Pipeline

The next section describes the SSAO pipeline. Its diagram is presented in Figure 3.4. Once the normal-depth buffer is generated, it is passed as an input to the ambient occlusion step. This stage is represented by two render passes, one for darkening factor computing and another for blurring the output. The normals obtained from the texture are in view space, so the coordinates of the sample directions are in fact the screen-space displacements of the sample points from the origin. As the vector is currently in view space, it has to be transformed to clip space first with the projection matrix uniform. Once the variable is transformed the perspective division has to be performed. The resulting coordinates are then transformed to the range $[0.0, 1.0]$ and used to sample the position texture.

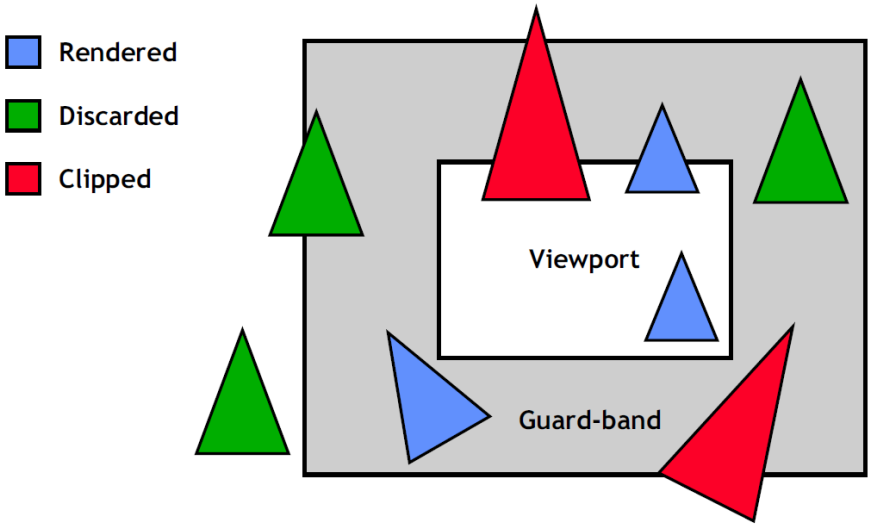

Figure 3.3: Guard-Band Clipping technique example from [11].

As for the rotation of the sample kernel, a random rotation vector could be created for every fragment. But on the other hand it increases memory consumption significantly, so a small texture of random rotation vectors is generated. In this implementation, it is represented as a [5,5] array tiled over the screen with the GL_REPEAT wrapping method.

Finally, the scene geometry is rendered again in the pass which writes directly to the screen framebuffer. It is responsible for accumulating of illumination and generating the final image displayed on screen. In this pass, previously created ambient occlusion 2D texture is used to attenuate the ambient term of the rendering equation. However, since SSAO map computes only a single value and not a vector, it can just be stored in the depth buffer instead of passing it to an additional texture.

To provide the user with variability when working with the application, four kernels of different sizes are created, which correspond to 'low', 'medium', 'high' and 'ultra' sampling quality settings.

This method requires following parameters: sampling quality, the sampling radius, the weight of occlusion, the depth constraint. These values are passed to the shaders as uniform variables. Using the provided interface, user can change the parameters for a given scene in order to get desireable visual result.

3.3 Horizon-based Ambient Occlusion

3.3.1 Description and implementation

The second technique that is supported in VEROSIM is Horizon-based ambient occlusion (HBAO). This method is another approximation of occlusion that was presented by NVIDIA [12]. This approach assumes the neighborhood of a point as a continuous

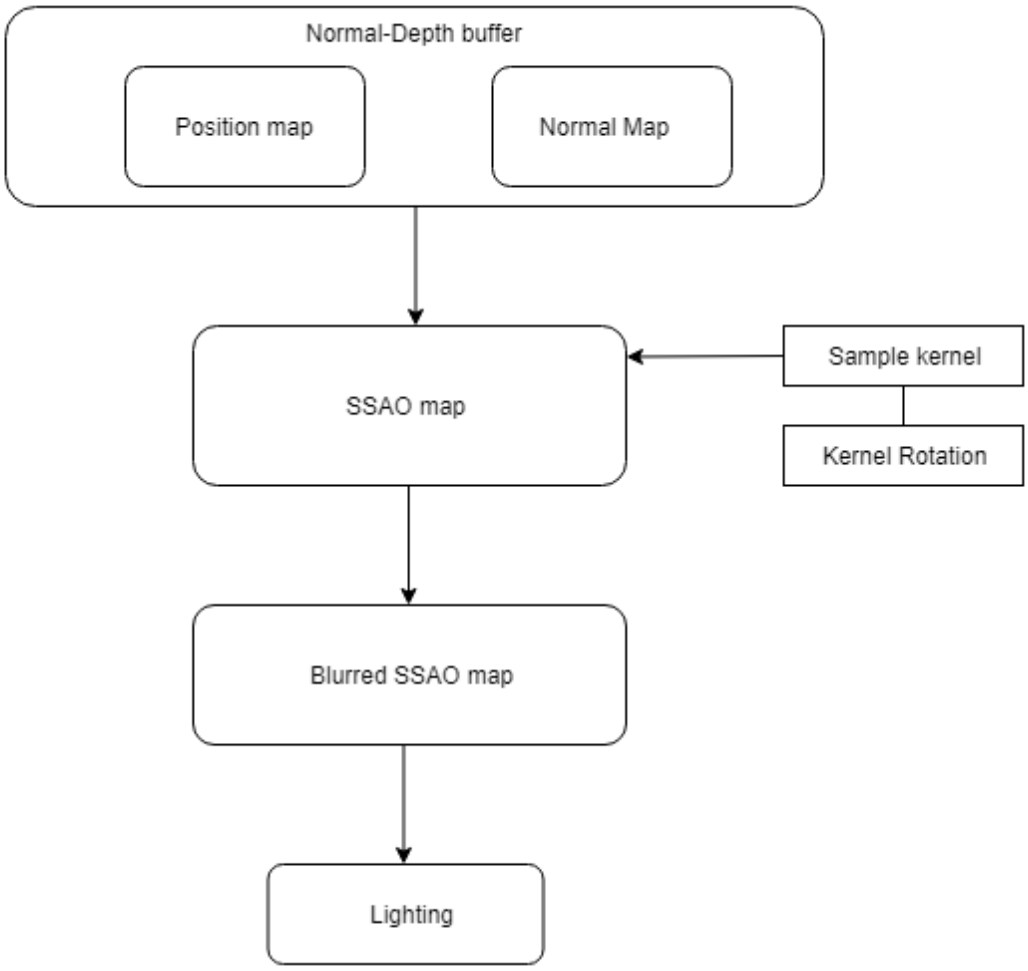

Figure 3.4: SSAO Rendering pipeline.

heightfield. In such case, if we trace a horizon line from a point in some direction ω then rays below the horizon are known to be occluded. Therefore, the intersection test of these rays can be dropped out without any evaluation of occlusion. The equation of ocllusion is:

$$A = 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} \int_{\alpha=t(\theta)}^{h(\theta)} W(\omega) cos(\alpha) d\alpha d\theta, \tag{3.1}$$

where W is linear function that is used for attenuation. θ stands for azimuth angle, α is the elevation angle (see Figure 3.5).

Then, the authors eliminate the double integral reducing the equation to:

$$A = 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} (sin(h(\theta)) - sin(t(\theta))) W(\theta) d\theta, \tag{3.2}$$

The attenuation factor $W(\theta)$ is taken equal to $max(0, 1 - r(\theta)/R)$, where R is the sample radius and $r(\theta)$ is distance between point P and the horizon point in deirection ω.

For computational purposes, Monte-Carlo numerical integration method can be used to solve the integral.

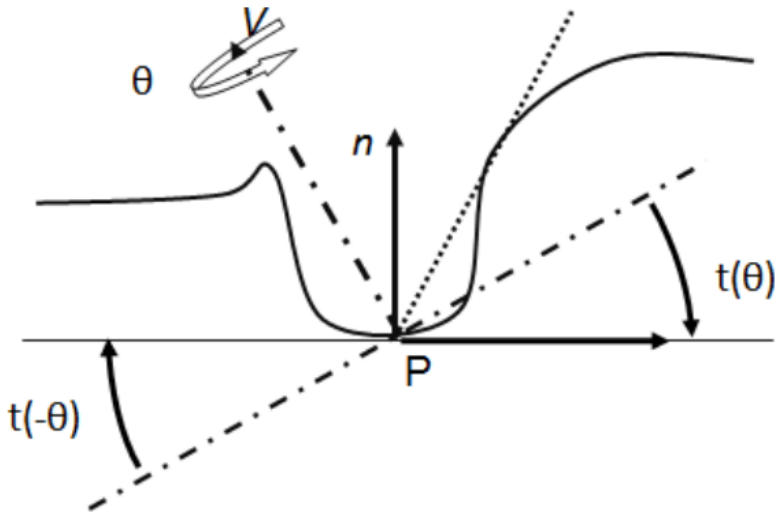

Figure 3.5: 'The azimuthal angle θ around the view vector. The tangent angle $t(\theta)$ is the signed elevation angle of the surface tangent vector.' [12]

HBAO is calculated with the ray-marching [13] algorithm applied to the depth buffer. The initial ray is directed perpendicularly to the view direction. A number of samples are gradually taken along this ray and the depth value is stored. If this the value decreases relative to the previous one, then we need to update the horizon angle. So every time the algorithm gets a point that is closer to the camera than the current nearest point, it is treated as an occluder and considered by the ambient occlusion function. The points which do not pass this comparison test are skipped, because they are occluded by the current closest point. Eventually every individual ambient occlusion factors for each sample are accumulated to represent the final occlusion of the point.

Previously, in SSAO the random rotation vectors were used to provide the high quality of the image and prevent artifacts. Horizon-based ambient occlusion also inherits this idea, but now it transfers to the randomization ray directions for each pixel. However, the relatively low number of samples and randomizing yields to dispersion which causes noise in the output image. Consequently, blurring needs to be performed again as a post-processing step.

In VEROSIM the algorithm is implemented inside the `renderHBAO()` function. It generates four sets of rays that are used for different sampling quality and set of sample directions that are rotated inside the shader. Then, the function passes these variables to the fragment shader as uniforms along with the following important parameters:

occlusion weight, sample radius, maximal step per ray. So most of the computation is done inside the shader. The final step is to smooth the result. This is done in the same way as in the SSAO algorithm.

The result of the SSAO and HBAO algorithms is presented in the Figure 3.6.

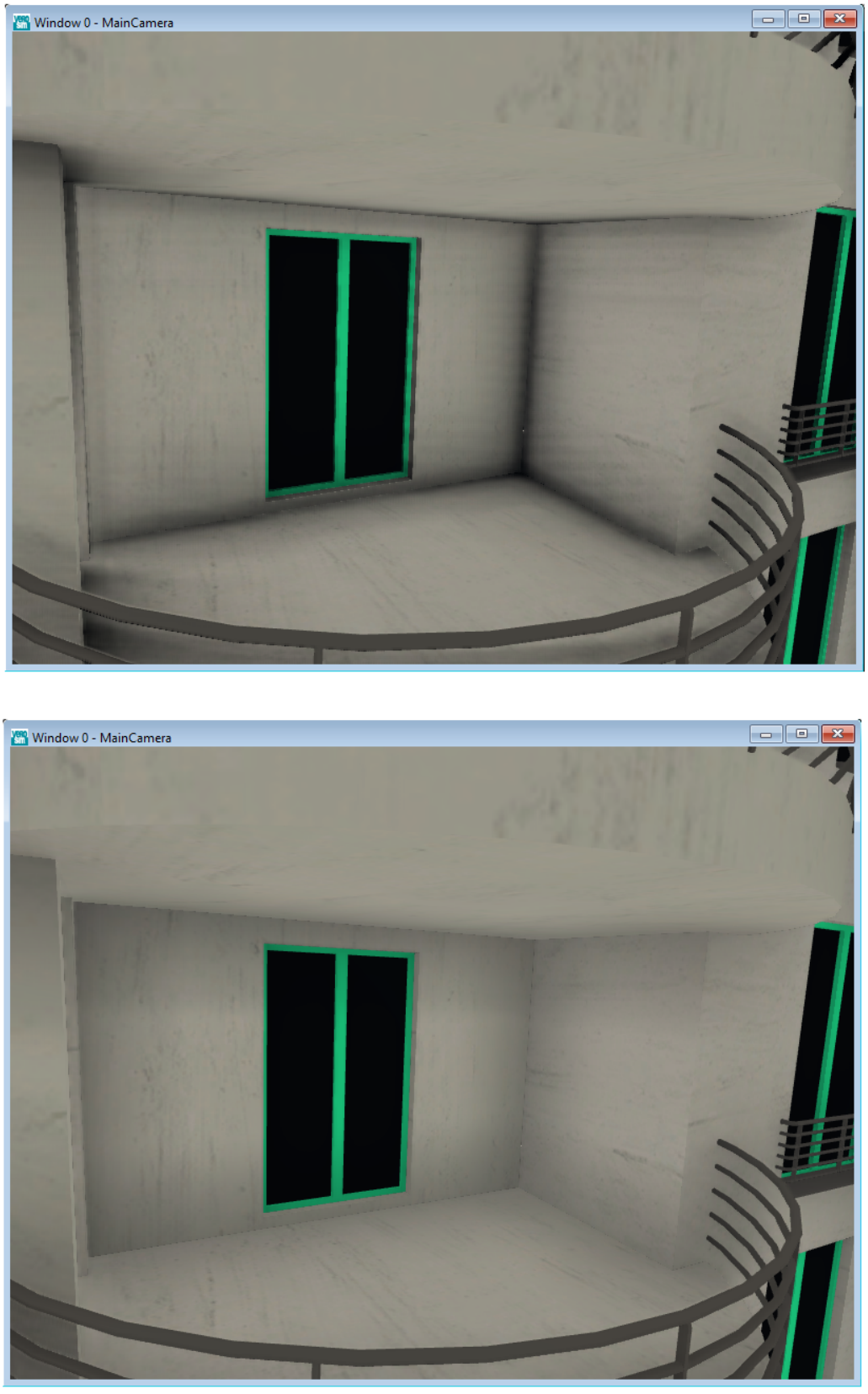

Figure 3.6: SSAO (top) and HBAO (bottom)

CHAPTER 4

SCREEN-SPACE DIRECTIONAL OCCLUSION

Now we proceed to the discussion of the Screen Space directional occlusion(SSDO) method. SSDO is essentially a generalization of the SSAO, in which the directional light is taken into account to calculate the shading. It is also opens up the possibility to handle one indirectional bounce of light [14]. In order to compute light transport, SSDO uses the normals and positions in three-dimensional space of each pixel in the screen space as input. The output result is generated in two stages. The first pass computes the direct illumination. The second stage includes using data from the previous step to acquire indirect bounces of light.

Since this algorithm is designed for screen space processing, the geometrical complexity of the scene should not affect its performance. Furthermore, resulting effects of occlusion and indirect illumination work with complicated dynamic scenes in real-time.

4.1 Direct lighting

According to standard screen-space ambient occlusion methods where only the 3D position and normal of every pixel is used, screen-space directional occlusion also takes the direction of the incoming light into account.

Denoting L_i as the incoming radiance from direction ω in the hemisphere, $\frac{\rho}{\pi}$ as the diffuse BRDF and V as the visibility test we can define the directional light equation. Total amount of light should be computed in the following way for each point x and N sampling directions:

$$L_{dir}(x) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_i(\omega_i) V(\omega_i) cos\Theta_i \Delta\omega \tag{4.1}$$

In the general case, the sum of sampling in the Monte Carlo method is represented by an integral over the hemisphere Ω:

$$L_{dir}(x) = \frac{1}{\pi} \int_{\Omega} L_i(\omega_i) V(\omega_i) cos\Theta_i d\omega \tag{4.2}$$

This method assumes that the incoming radiance L_i can be efficiently computed from environment maps, directional or point lights. Similarly to the screen-space ambient occlusion method, we do not use any ray-tracing technique and approximate the occluders in image space. However, a slight difference lies in that SSDO uses sampling points generated in the hemisphere in three-dimensional space. This is done by using pixel 3D position and its normal, in contrast to screen-space ambient occlusion where the samples were distributed over the flat disc in the image space.

The following steps are performed to sample of the hemisphere: For every generated direction ω_i and a random step $r_i \in [0, r_{max}]$, where r_{max} is the user-defined radius. The position of the sampling points is computed as $x + r_i$. The generated points are located in the corresponding hemisphere centered at x in 3D. This hemisphere is oriented around the normal n. Then the backprojected sampling points depths are compared with the values stored in the original z-buffer. If the depth of the sampling point is greater than the depth value from the original z-buffer, this tells us that the sampling point is located under the surface. Thus, the visibility test is failed since the light from this particular direction is blocked by an occluder. Otherwise, the light reaches a point in this direction without meeting any obstacles and the incoming radiance L_i can be calculated.

In Figure 4.1 we observe example with four sampling points A, B, C, D. These sampling points are randomly generated with a uniform distribution over the hemisphere with a random step from the original point X. Then sampling points are backprojected on the surface. Now since we have information about the coordinates in image space, we can get the 3D coordinates from a frame buffer. These points are again projected back into the image in order to get the distance from the camera position. If the sampling point is further than the appropriate point on a surface, the sampling point is classified as an occluder. In our case points A, B and D act as occluders because their projection is moving towards the eye. Otherwise, the projection of sample C falls in the opposite direction, so only this point participates in the final illumination computation.

4.2 Indirect bounce

Using the preceding step we can now simulate the indirect lighting pass. Since the normals and 3D coordinates are known for all projected sampling points, they can be used to obtain one indirect bounce of light. The bounce algorithm considers only the surface's points which were projected from sampling points and treated as occluders. For each of these pixels the directional light intensity L_{dir} and the corresponding pixel color from the direct illumination form the basis for indirect light. In accordance with this information, the indirect radiance being sent in the direction of the point x can be calculated. Pixels are now treated as tiny patches around their normals. Using the sender normal helps to cope with the problem of noise artifacts such as color bleeding

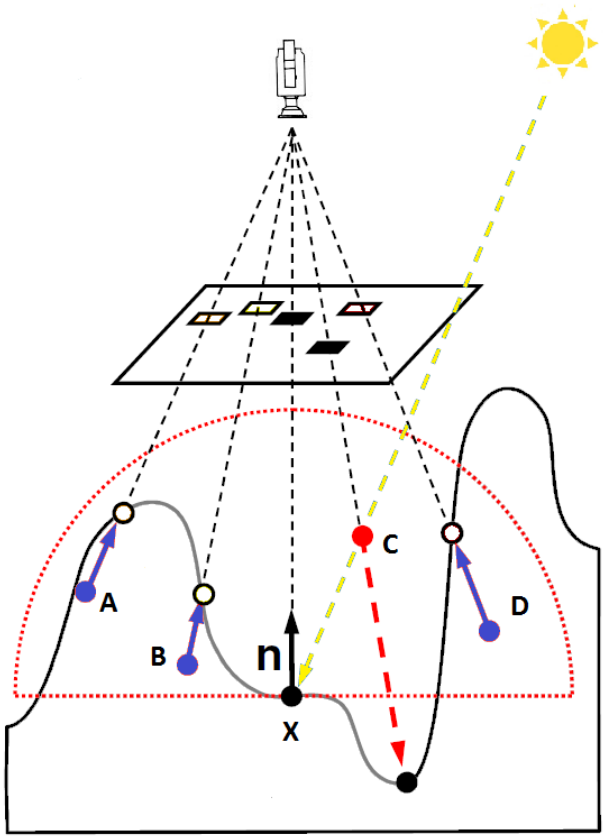

Figure 4.1: Four random samples are represented in the generated hemisphere. Each sample undergoes the occlusion test. Here only the sample C successfully passes it [14].

from back facing patches. The equation for the additional incoming indirect radiance for a point x:

$$L_{ind}(x) = \sum_{n=1}^{N} \frac{1}{\pi} L_i(1 - V(\omega_i)) \frac{\pi r_{max}^2 cos\Theta_{s_i} cos\Theta_{r_i}}{N d_i^2}, \tag{4.3}$$

where Θ_{s_i} is the angle between the direction and sending normal, Θ_{r_i} represents analogical relation regarding the receiving normal. The distance between the point x and sampling point is denoted as d_i. The area of the sender patch in this equation is defined by fraction $\frac{\pi r_{max}^2}{N}$. This value can be considered as a parameter for manual control of the color bleeding strength.

In figure 4.2, only three out of the four samples are occluders, but the indirect bounce is considered only for samples B and D because the normals of their projections onto the surface are directed towards point x. While the normal from the patch related to the point A is directed in the opposite direction, it will not contribute to the final bounce.

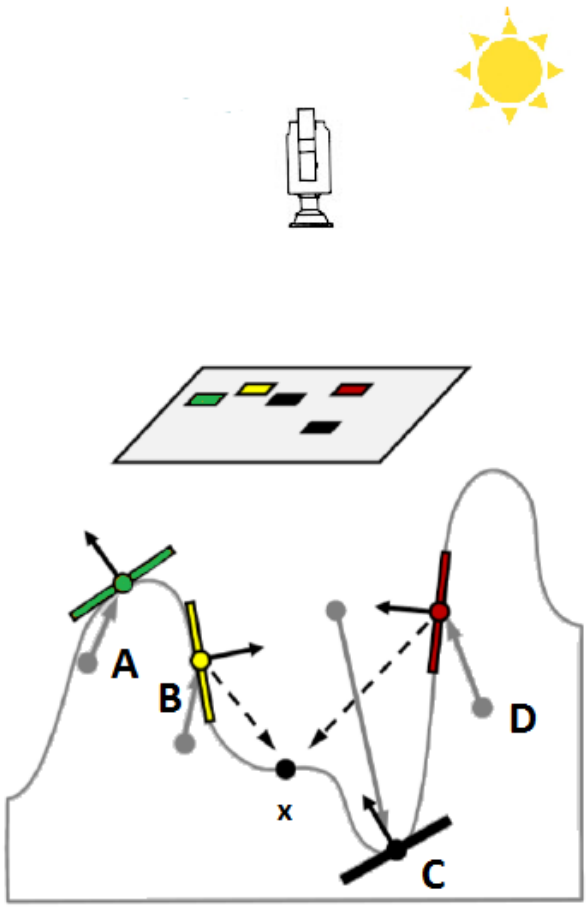

Figure 4.2: Indirect light calculation. A small patch is placed on the surface for each occluder and the direct light stored in the framebuffer is used as sender radiance.

4.3 Smoothing

4.3.1 Gaussian blur

Directional occlusion provides more detailed perception of the scene depth. However, without filtering it produces the visible noise in the texture and this applies to both direct lighting and indirect bounce passes of the algorithm due to random sampling. To overcome this issue it is necessary to apply an advanced filtering technique.

Under filtering signifies averaging the value of neighboring pixels. But if we consider the uniform distribution and a simple arithmetic mean, the blurring result is still coarse. Therefore, the neighbor pixels have to be multiplied by coefficients that obey the normal distribution law (Gaussian distribution). These coefficients can be found from formula:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{4.4}$$

where μ is the expectation and σ^2 is variance.

Gaussian blurring has one important property - separability. This makes it possible to divide the algorithm into two parts - blurring along the x coordinate and blurring along y coordinate respectively. This separation reduces the complexity from $O(N^2)$ to $O(N)$, allowing you to increase the performance significantly. The Gaussian blur produces smooth results by representing each pixel as a function of both the color value of neighbouring pixels and their distance to the center of the kernel. An example of blur is shown in the figure 4.3.

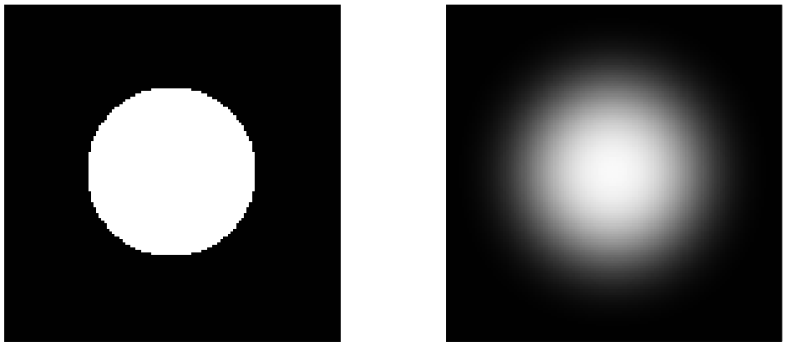

Figure 4.3: Original image(left) and its filtering with the Gaussian blur(right).

4.3.2 Bilateral filtering

The inaccuracy of Gaussian blur lies in the fact that it does not take in account the geometry of a scene. So it is insensitive to the borders of different objects. Consequently, the boundaries of objects are not respected and blurred away. The bilateral filter extends the standard Gaussian blur definition to cope with these shortcomings. It is taking into account not only the distance to its neighbours and coefficient value, but also checks the difference between pixel intensities. For some image I the equation for this filtering $B(I)_p$ is:

$$B(I)_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(||p - q||) G_{\sigma_i}(|I_q - I_p|) I_q, \tag{4.5}$$

where p is the main pixel, q - the pixel from neighborhood S, so the distance between them is written as a norm $||p - q||$. W_p is the normalization factor. I_p stands for pixels intensity. σ_s and σ_i are parameters for space and intensity weights. The bilateral filter can be controlled by these two parameters. Increasing σ_i, the filter becomes closer to Gaussian blur. The increase in the spatial parameter σ_s smoothes bigger features.

Although the authors of the original SSDO algorithm suggest using a different geometry-aware blur [15], we make our choice in favor of the bilateral filter [16]. When

this filter is used for directional occlusion, the aim is to blur the output of the shader to remove noise that appears because of random sampling. To perform it, the second weight term converts to a function of pixels depth istead of intensity.

Figure 4.4: A pictuire before(left) and after(right) bilateral filtering[16].

Although the bilateral filter provides better results by handling the geometry of a scene, its performance is relatively slow. The filter has to work with a number of pixels that equals to the square of the kernel's size. According to this, the complexity of the algorithm is $O(N^2)$. This can be crucial in context of real-time rendering within a highly loaded system. So we need to speed this step up, preferably without loss of quality.

The idea is again to separate the steps in two passes: one for vertical blurring and another for horizontal. Turning to mathematical definitions, the image is first blurred only with respect to the x-axis. It is performed in the same way as it was done earlier, with the only change that the new neighbourhood of the point is represented as a line instead of a circle:

$$I_{p_x} = \frac{1}{W_p} \sum_{q \in X} G_{\sigma_s}(||p - q||) G_{\sigma_i}(|Depth_q - Depth_p|) I_q, \tag{4.6}$$

In this intermediate step, the summation is carried out only for the line of pixels that is alligned to x-axis neighborhood X with center in point p.

The second step is not much different from the previous one, but requires the I_{p_x} for calculation and performed with respect to y-axis neighborhood Y:

$$I_{p_y} = \frac{1}{W_p} \sum_{q \in Y} G_{\sigma_s}(||p - q||) G_{\sigma_i}(|Depth_q - Depth_p|) I_{p_x}, \tag{4.7}$$

The longer formulation however decreases complexity. With N number of pixels it is now equal to $O(N)$. However, it is important to note that technically this filtering is non-separable. But pursuing the real-time performance, this turns out to be a fair approximation to the existing algorithm in practice.

Figure 4.5: SSAO map smoothed with bilateral filter.

4.4 Limitations

Performing the visibility test as it is described in section 4.1 and 4.2 may cause some misclassification of the sampling points. This can happen due to the fact that we operate with only one depth value and do not have the information about occluded objects. As shown in Figure 4.5 we can miss a space of incoming light or treat a blocked direction as visible. An issue with sample B can be resolved by adding a higher number of samples in this direction. However, the gap at sample A can not be discovered from a single viewing position because we do not store any information about the geometry behind the first depth value.

A possible solution to overcome this problem is called *depth peeling*[17]. For every pixel, multiple depth values are sent to the frame-buffer. With depth data, the occlusion test can be improved as we gain better understanding of the scene geometry. Now if the sampling point is behind the first depth value, we need to check its position with respect to the second depth. These z-buffer values belong to back and front faces of the object. So if the sample is located between two depths, this means that it is inside the object.

Another approach to detect hidden regions is the use of multiple cameras. With

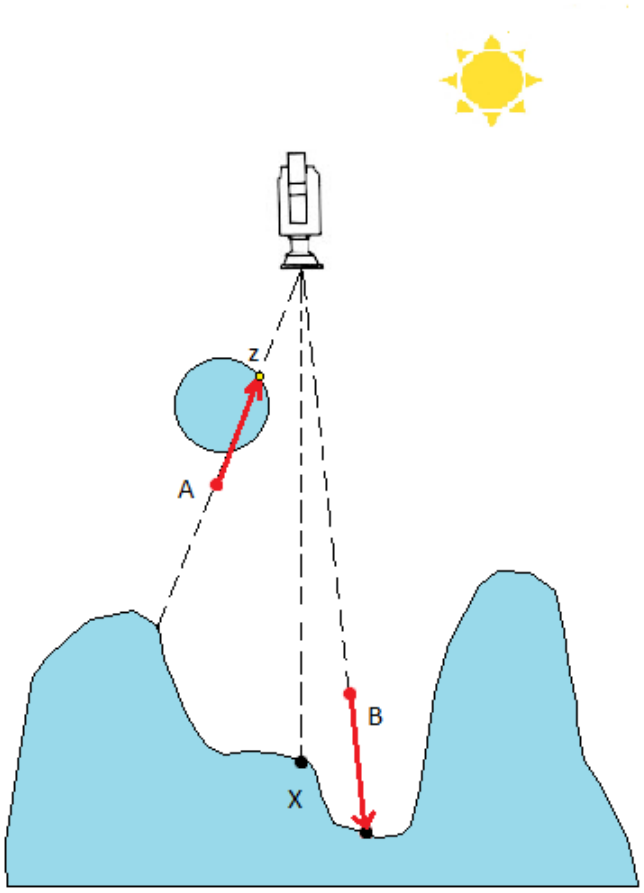

Figure 4.6: Possible cases of incorrect illumination. Sample A is visible but its projected point is closer to the camera, so the algorithm determines it as an occluder. Although sample B is above the surface, the direction from its projection is blocked by the surface.

different viewing positions we can get the information about off-screen obstacles. Additionally, different camera positions can improve the processing of the faces that are viewed at small angles. These tilted areas cannot be correctly handled. As a result, color bleeding from them disappears. But when an additional camera is used, such sources of indirect illumination become noticeable. A good idea for positioning an additional camera is to rotate it 90 degrees relative to the main camera around the object. It turns the previously tilted face so that it becomes more visible. However, this may lead to obstacle problem: part of the objects may be blocked for an additional camera with respect to the main view. In this case the depth peeling can be applied for a second camera as well.

Both depth peeling and multiple cameras require more computing power. According to [14], the overhead is up to 160%. Taking into account the conditions of the primary task, which was to achieve smooth operation of the algorithm within a highly loaded

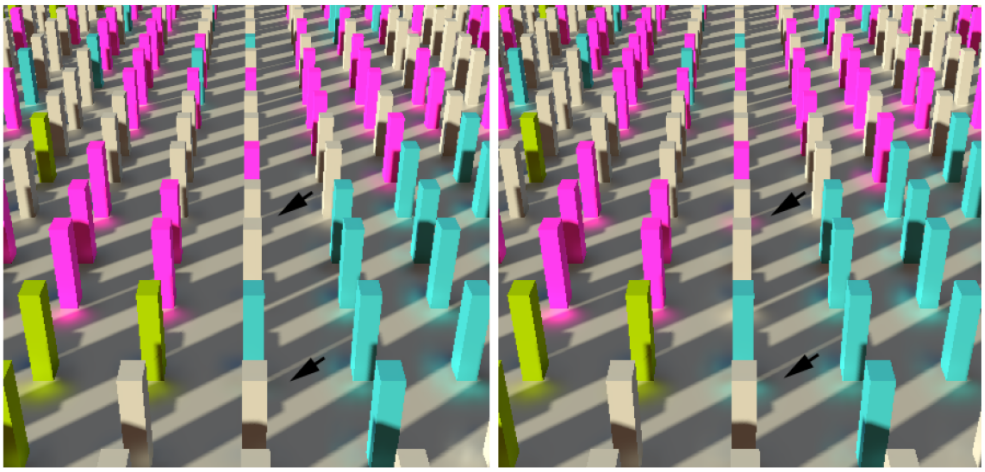

Figure 4.7: Using of extra cameras (right) enables the contribution of light to unoccluded regions, that was not possible with the single camera [14].

framework, these improvements were not implemented and embeded for the sake of performance.

4.5 Model implementation

For initial tests, a simplified prototype of the indirect lighting model is implemented. This prototype is developed in Microsoft Visual Studio IDE[1] using C++ and Open Graphics Library (OpenGL)[2]. Accordingly, all shaders are written in GLSL. It should be noted that the VEROSIM framework uses the same technologies, which facilitates further integration. The OpenGL Extension Wrangler and GL Frame Work libraries are also used for windows managing and input device handling. GLM library is responsible for mathematics (operations over vectors and matrices).

The program contains the following sequential steps:

1) *Geometry pass*: During this step we load the model and store the fragment position vector in the first g-buffer texture. Then the depth value is stored in the same four-dimensional vector as alpha component. The fragment normals are sent into the g-buffer that corresponds to the normal layout.

2) *Indirect light pass* Input parameters for this pass are the normal texture, the position and the depth of our g-buffer. For each kernel sample it runs the bounce test. If test is passed successfully we accumulate the bounce value in the *indirectLight* variable. The output of this pass is demonstrated in Figure 4.8.

3) *Indirect light pass blur*: Now we need to reduce the sampling noise by filtering. This pass takes the output of the previous pass as input and blurs the value of each fragment with its neighbors. The output is saved separately. The result is in figure

[1]https://visualstudio.microsoft.com
[2]https://www.opengl.org/

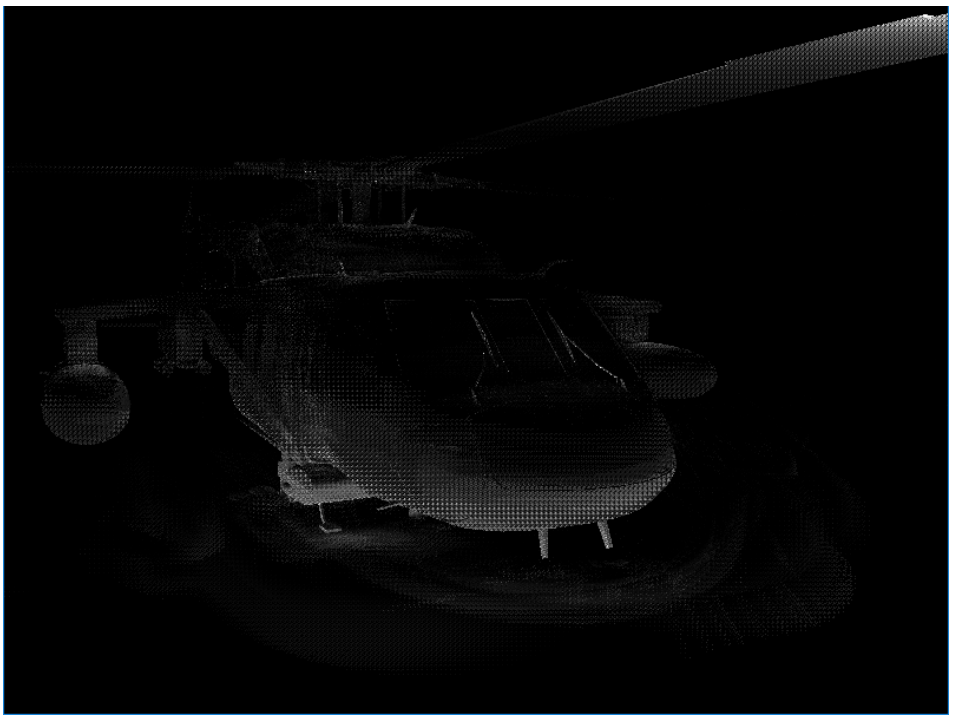

Figure 4.8: The unblurred indirect light bounce output. Increased value of the intensity is chosen to emphasize the effect

4.9.

4) *Combination with the direct light.* Finally, we mix the blurred image with the sum of the diffuse and specular Blinn-Phong values of a point light, and the directional light generated by the environment cube map.

4.6 Integration into VEROSIM

We extend VSPluginRenderGLAmbientOcclusion plugin to support the above rendering algorithms and features.

The same sampling algorithm as in the SSAO is used. This allows to compare rendering methods more objectively later. The samples are interpolated in such a way that most part of them are located closer to the original point. This is done in order to obtain larger weight on occlusions close to the actual fragment [18]. Figure 4.11 shows this point distribution.

The function `renderSSDO()` is defined for both directional occlusion and indirect bounce passes. Switching between modes is triggered by boolean variable *indirect*. The function passes all needed parameters to the appropriate shader. This shader generates a directional occlusion map. In contrast to the grayscale texture of the SSAO map, the SSDO passes require a colored one. Therefore, we initialize the texture buffers with relevant parameters.

Figure 4.9: The same helicopter model after blurring

Figure 4.10: Direct lighting + indirect bounce

Further calculation of the indirect bounce can be performed either inside the previous shader or in its own. In the first case, no additional sampling is required and the

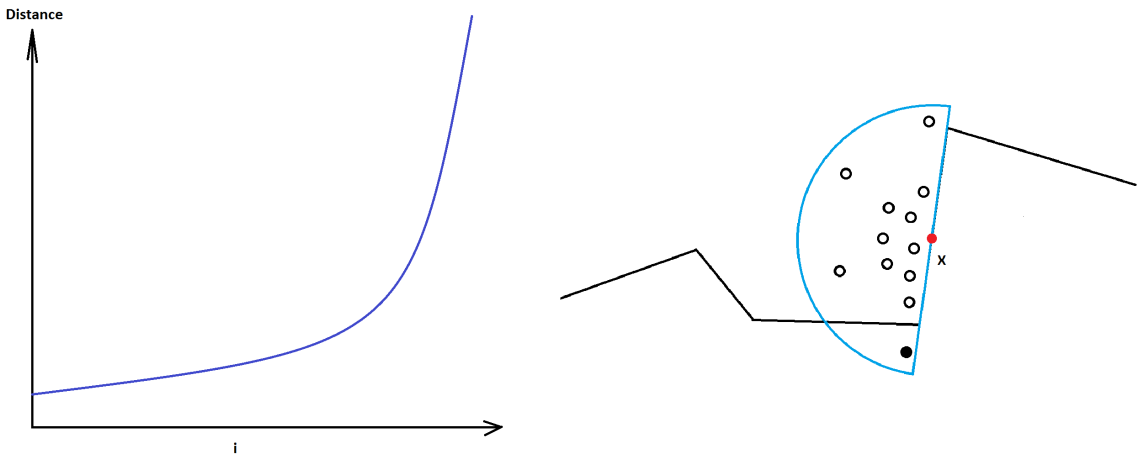

Figure 4.11: The dependence of the number of samples on the distance from the origin(left). Corrected sample distribution(right).

same samples are used for both passes. This reduces the number of additional function calls, which has a positive effect on performance. On the other hand, different sampling could theoretically have a positive effect on quality. However, in practice, the output texture is indistinguishable to the human eye, while the performance drops. Both approaches are implemented in VEROSIM, but the first one is taken as the basis algorithm. The maps created by ssdo shader are presented in Figure 4.12.

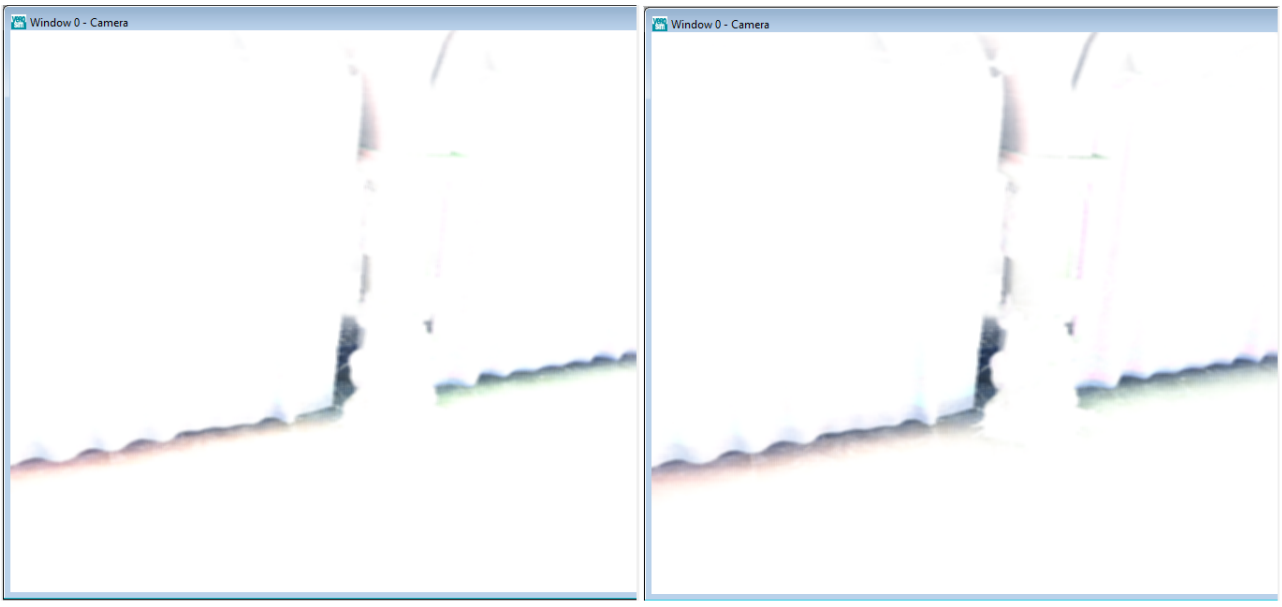

Figure 4.12: Screen space directional occlusion intermediate steps. Crytek Sponza model. Light bounce (left). Directional occlusion (right) produces colored shadows.

A similar situation occurs with filtering. In the case when the program operates with two different shaders, the blur can be applied to them separately. However, nothing interferes to smooth the combination of passes only once.

Eventually, the results of all passes are collected in the *'combineSSDO'* shader, which generates the final texture that is displayed on the screen(Figure 4.13).

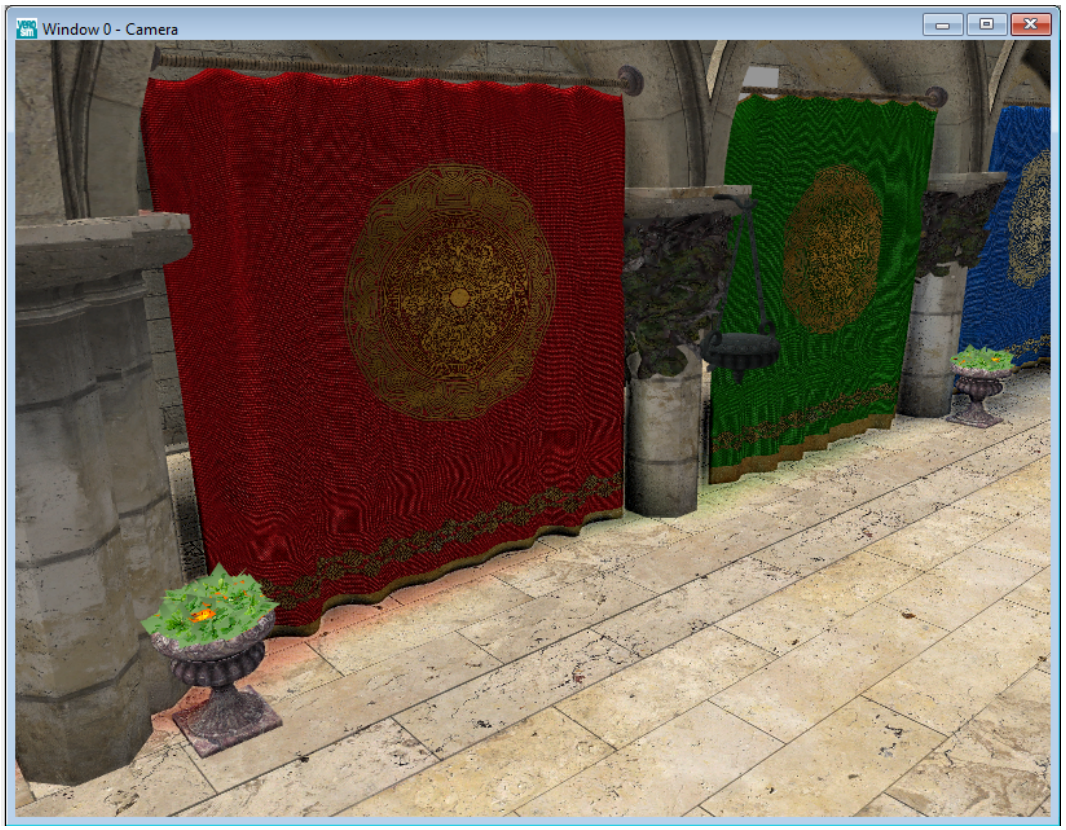

Figure 4.13: SSDO+indirect bounce.

The updated interface of the plugin is presented in Figure 4.14.

4.7 Possible improvements

4.7.1 Subsampling

Although the original algorithm does not use any subsampling, this kind of discretization can be applied to it under certain assumptions. To perform it efficiently regarding image quality, the indirect lighting should change slowly throughout the scene. If this is the case, the pass of indirect light can be subsampled in order to increase the performance. Depending on the subsampling size, the number of fragments processed inside the shader decreases quadratically with respect to it.

The same improvement applies to the filtering. Subsampling lowers down the resolution of image to be blurred. As a result, the bluring stage also speeds up.

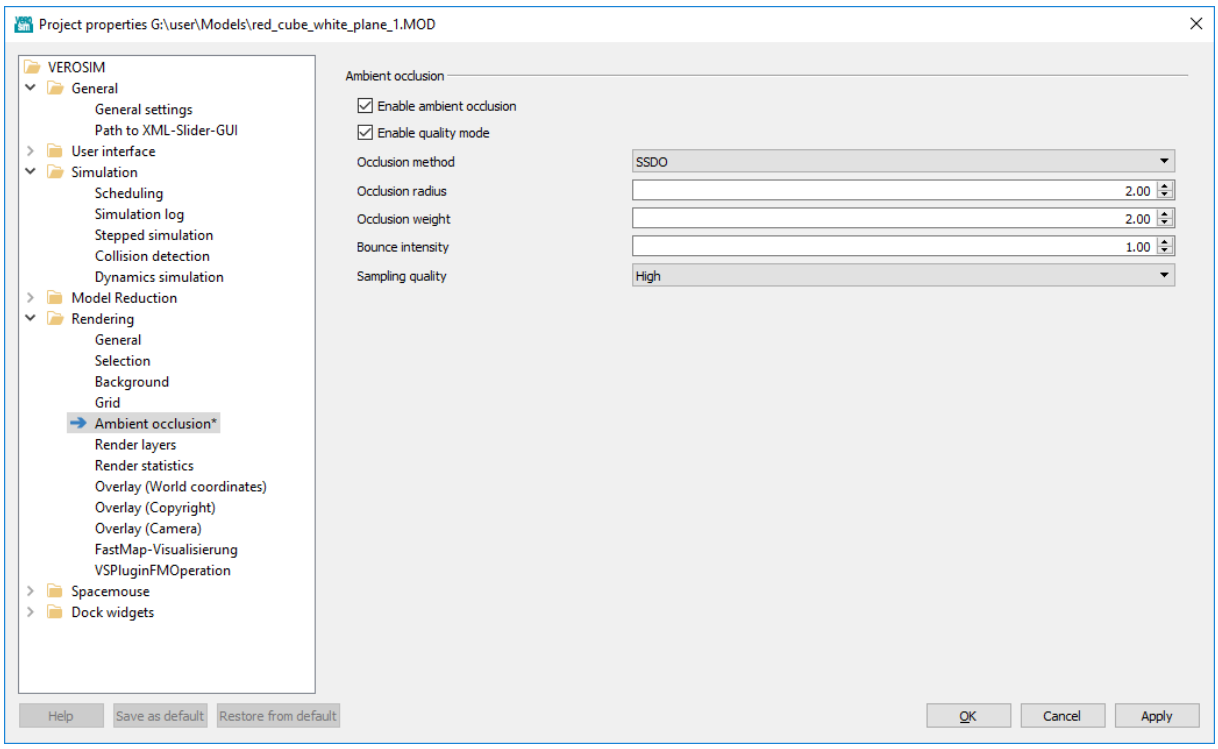

Figure 4.14: Along with the weight, occlusion radius and sampling quality parameters, the user can now choose two additional methods "SSDO" and "SSDO + bounce". The bounce value also can be manually adjusted.

However, a disadvantage of the subsample is that the calculated values require the correct upsampling at the end. So the scene geometry features must be considered, because objects can lose their fine detailes.

4.7.2 Filtering

We use a bilateral filter as an improved version of Gaussian blur. However, it also may have errors and can create undesirable noise. The information about geometry is important again to prevent color bleeding over edges. This is especially true for full dynamic scenes with a large number of moving objects. For such cases, it makes sense to use specially adapted techniques. For example feature-aware filter presented in [19]. It is a post-process blur filter. The method is designed to be robust regarding scenes with complicated inter-object motions and fine-scale details.

CHAPTER 5
EVALUATION

5.1 VEROSIM Rendering profiler

To evaluate the performance of the presented methods, we use the *VSLibRenderGLProfiler*, which is part of the VEROSIM framework. To integrate it into the configuration, the following plugins are required: VSLibRenderGL, VSPluginRenderGL and VSLibRenderGLProfiler.

The first thing to do is to start a scene simulation to ensure continuous rendering. Afterwards the action VSPluginRenderGL.EnableProfiler can be triggered to activate the profiler. After the analyzed part of the simulation is completed, the profiler can be turned off and its results can be can be observed in the viewport. It is represented in hierarchical tree format. The profiler has tabs to track the load on the processor and graphics card. Since a significant part of the calculations in the tested methods is performed on a video card, the time spent on the CPU load can be neglected. The Figure 5.1 shows the profiler interface.

The average time in milliseconds is selected as the metric for the evaluation.

The tests were preformed on the PC with following chracteristics.

- NVIDIA GeForce GTX 960 graphics card
- Intel® Core™ i7-940 CPU 2,93 GHz, 2,93 GHz
- 12 GB RAM

5.1.1 Performance and visual results

The main scene used for tests is 'Halifaxstrasse.MOD'. General view of the scene is shown in Figure 1.1.

Every algorithm (SSAO, HBAO, SSDO, SSDO+bounce) was run several times with following varying parameters: Sample kernel size N (12, 24, 32, 64), occlusion radius r (2, 4, 8). These parameters directly affect performance, in contrast to occlusion weight, which is just a darkening factor. The resolution in every test is set to 800x600.

SSAO	**r = 2**	**r = 4**	**r = 8**
N = 12	0,78	1,03	1,2
N = 24	1,14	1,61	2
N = 36	1,36	2	2,4
N = 64	2,28	3,46	4,3

HBAO	**r = 2**	**r = 4**	**r = 8**
N = 12	1,2	1,44	1,54
N = 24	1,7	2,16	2,55
N = 36	2,14	2,8	3,35
N = 64	3,14	4,3	5,36

SSDO	**r = 2**	**r = 4**	**r = 8**
N = 12	1,6	2,8	2,5
N = 24	3,2	4	4,9
N = 36	4,5	5,1	6,52
N = 64	8,2	10,4	11,8

SSDO+	**r = 2**	**r = 4**	**r = 8**
N = 12	1,9	2,5	3,08
N = 24	3,8	5	5,9
N = 36	5,1	6,7	7,9
N = 64	10,2	13,2	15,6

On the basis of the average values from the obtained tables, a graph(Figure 5.2) of the executing time versus the number of samples is plotted.

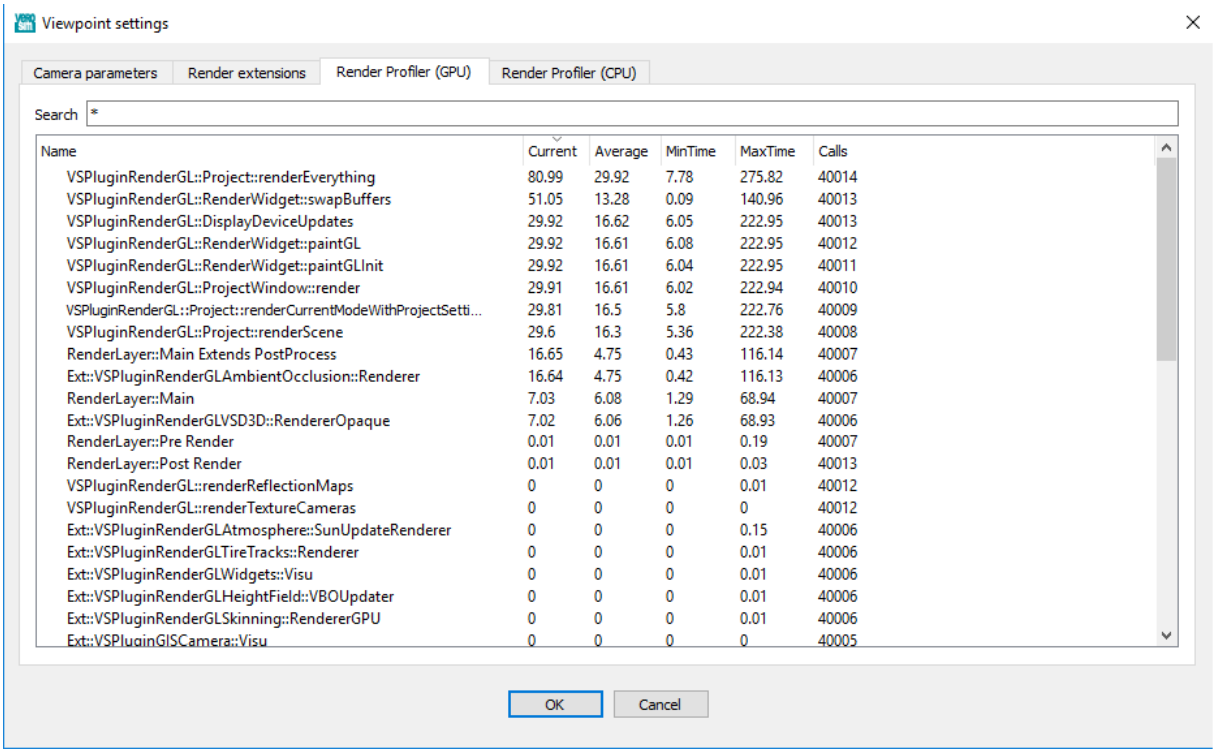

Figure 5.1: The profiler displays current, average, minimum and maximum time spent on call. The total number of calls is also available.

For low quality sampling, the difference in performance of the methods is relatively small. However, with an increase in the number of samples, the execution time of the screen-space directional occlusion and its modification with one indirect bounce increases noticeably.

The fastest method proved to be the standard SSAO. The overhead of the horizon-based approach is about 30% on average. The performance of the SSDO is about 20%

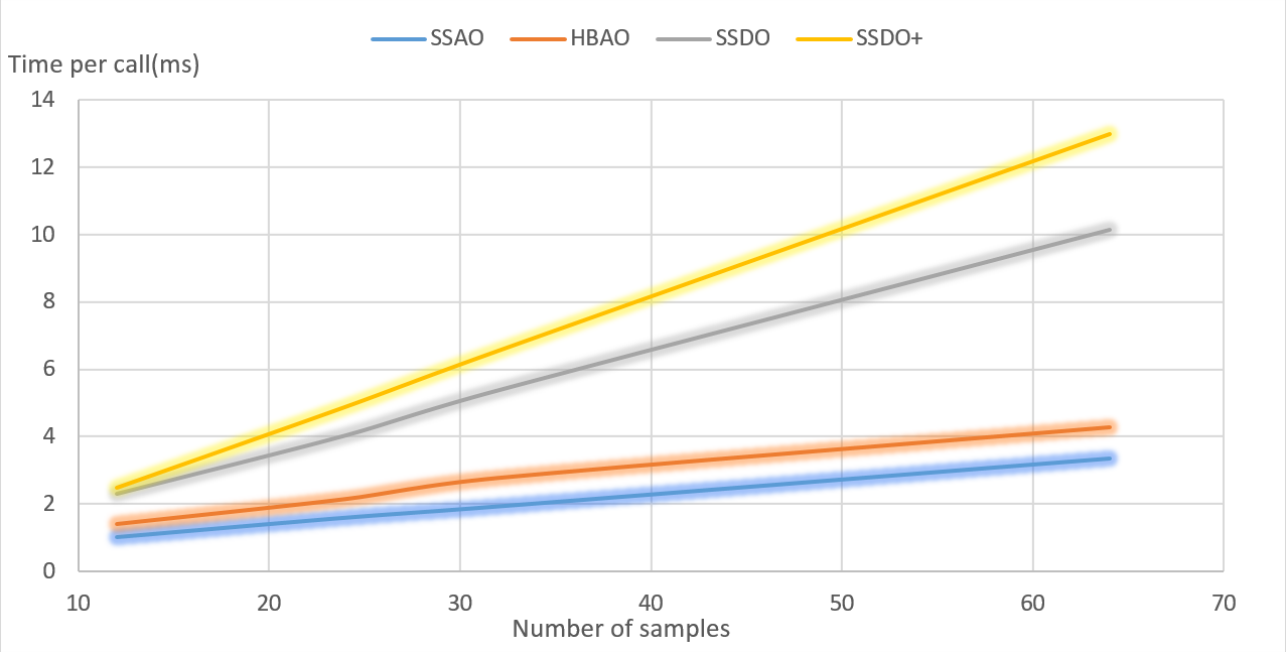

Figure 5.2: The processing time of SSDO and SSDO+ increases markedly with for big number of samples.

faster than SSDO+, which is consistent with the results in [14].

As for the frame rate, when any method of occlusion is disabled, the scene is stably rendered at 90 FPS. Enabling SSAO lowers it to 65 FPS which is still smooth. With HBAO, renderer produces about 60 frames per second on average. Directional occlusion algorithm works at 37 FPS. SSDO+ provides 33 FPS. The reduced performance of the last two algorithms seems to be due to different internal format of textures and increased set of parameters passed to the shaders. The number of shader calls also affect performance.

Figures 5.3 and 5.6 demonstrate the results of all four algorithms.

5.2 Head-mounted display

The VEROSIM framework includes support for VR headsets, in particular the HTC Vive. In this regard, it was decided to test the performance of the algorithms also in virtual reality environment.

The interaction between the head-mounted display and the system is carried out using the `VSPluginOpenVR` plugin. After installing the plugin, the option `Enable Stereo Headset` becomes available in the project settings. A rendering window is associated with every VR headset. It is able to display the VR view using following modes:

- None: The render window displays nothing.

Figure 5.3: Top left: SSAO, top right: HBAO, bottom left: SSDO, bottom right: SSDO+bounce.

- Mirror: The textures for the left and right eyes are mirrored on the screen.
- SinglePass: The scene is rendered in monoscopic mode but from the viewing angle of the VR headset.

The response time for the renderer is set to 11.11ms. That is exactly 90 frames per second, which is the standard for modern head-mounted displays. In case the system cannot maintain the necessary frame rate, the device is able to reproject the old frames in order to obtain a smooth simulation[20].

For tests with the HTC Vive, another PC with the following specifications was used:

- NVIDIA GeForce GTX 1080Ti graphics card
- Intel® Core™ i7-8700K CPU 3,7 GHz, 3,7 GHz
- 16 GB RAM

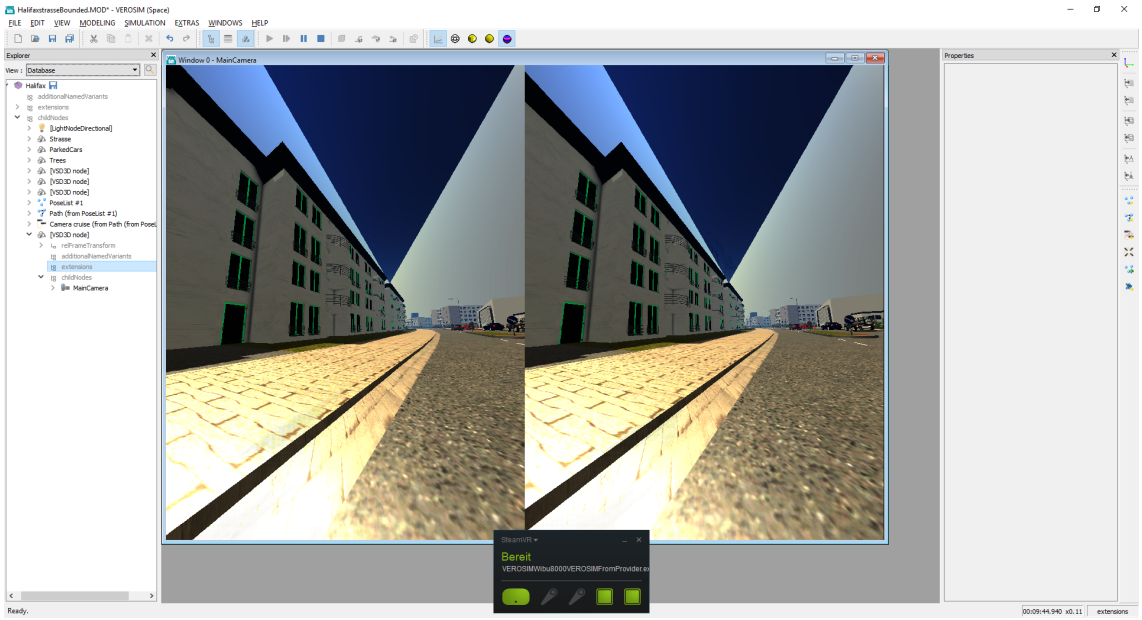

Figure 5.4: Stereo view mode in VEROSIM.

SSAO	r = 2	r = 4	r = 8
N = 12	1,7	1,8	1,7
N = 24	3	3,1	3,2
N = 36	4,2	4,3	4,6
N = 64	6,6	7,4	7,8

HBAO	r = 2	r = 4	r = 8
N = 12	2,03	2,3	2,49
N = 24	3,13	4,5	4,92
N = 36	4,15	6,4	6,9
N = 64	7	9,8	10,8

SSDO	r = 2	r = 4	r = 8
N = 12	7	7,36	8
N = 24	11	13,2	14,1
N = 36	16,6	17,7	18,2
N = 64	31	31,92	32,71

SSDO+	r = 2	r = 4	r = 8
N = 12	8,2	9,2	9,4
N = 24	14,9	17,9	18,1
N = 36	21,4	23,3	23,7
N = 64	37,7	41,1	43,7

It is important to note that the display resolution of the HTC Vive is 1080 x 1200 pixels for each eye. Therefore, longer call processing is expected. Apart from differences mentioned above, performed similar measurements using the rendering profiler and obtained the following results:

Tests showed similar behavior. However, for a large number of samples, the radius variation has a relatively small impact on performance. The visual perception of the scene in head-mounted display is smooth up to an increase in the number of samples to 64. The system also corrects the noise that may occur during normal work with the scene.

The graph of system performance is shown in the figure 5.5.

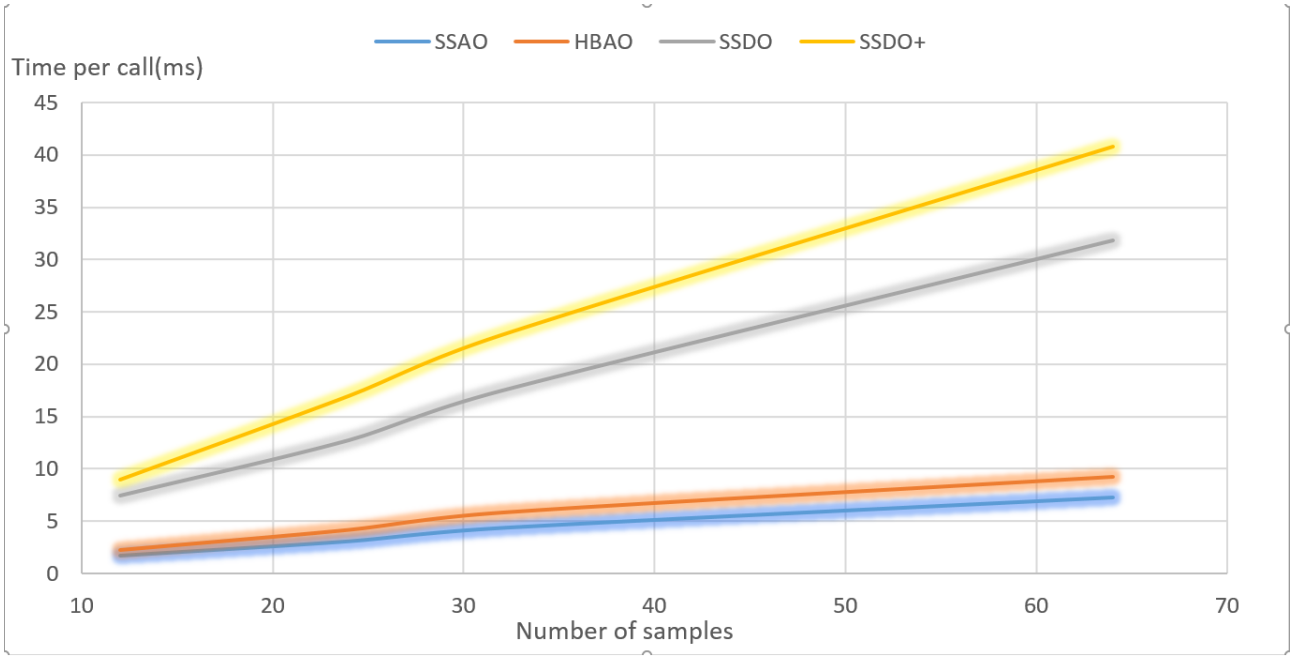

Figure 5.5: Processing time for VR.

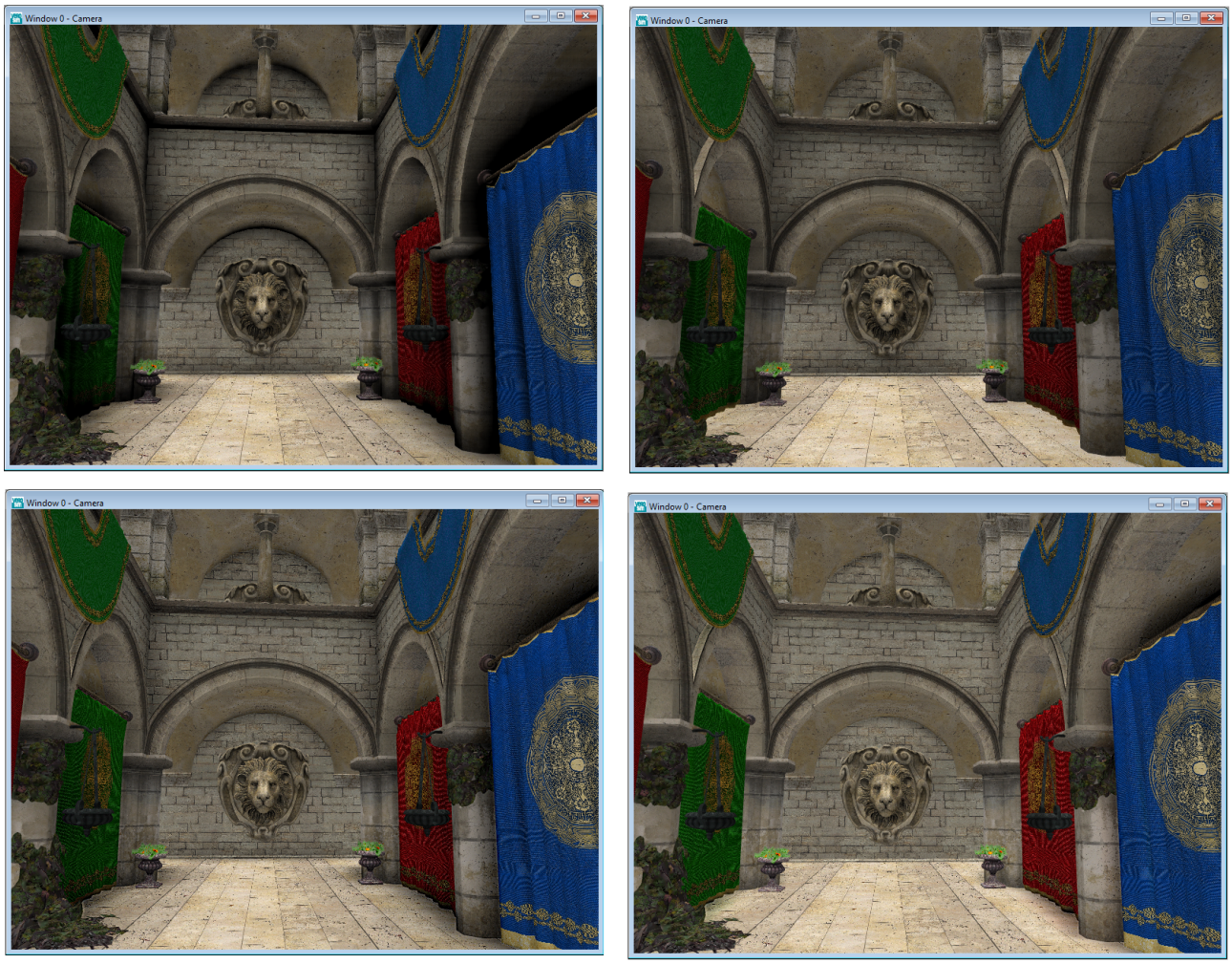

Figure 5.6: Top left: SSAO, top right: HBAO, bottom left: SSDO, bottom right: SSDO+bounce.

Chapter 6
Conclusion

In this work, we consider methods for global illumination approximation in screen space. In addition to the SSAO and HBAO approaches, the screen-space directional occlusion algorithm is implemented. SSDO is a generalization of the standard screen-space ambient occlusion. Method takes into account directional light, which allows to determine the occlusion factors more precisely. A simulation of one indirect bounce of light is also implemented in order to achieve a better lighting model. Using geometry-aware filtering is more efficient at smoothing artifacts and preventing unnatural color bleeding.

Additional features are integrated into the VEROSIM framework as an extension to the existing rendering plugin. Every occlusion method can be enabled during the simulation various scenarios, including dynamic scenes. The user can adjust the shading parameters manually.

SSDO and its modification showed slower performance relative to standard solutions. However, they calculate a more accurate result, which depends on the light position and objects color. All methods work in real-time, demonstrating a stable frame rate. Although it does not reach 90 frames per second under conditions of a highly loaded simulation system, the results can be considered as acceptable. In a virtual reality environment, the simulation also looks smooth thanks to the reprojection technique.

As further work, multi-depth buffer, multiple viewports and modern geometry-based blurring can be used to improve the accuracy. The subsampling technique can increase the performance and establish a fair balance between speed and quality.

Bibliography

[1] Nuno Subtil, Eric Werness: NVIDIA RTX: Enabling Ray Tracing in Vulkan, 2018. http://on-demand.gputechconf.com/gtc/2018/presentation/s8521-advanced-graphics-extensions-for-vulkan.pdf.

[2] J. Rossmann: Advanced Virtual Testbeds: Robotics Know How for Virtual Worlds. In: The 11th International Conference on Control, Automation, Robotics and Vision (ICARCV 2010). Singapore, 2010, pp. 133–138.

[3] Unreal Engine Lightmass Global Illumination. https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/Lightmass. [Retrieved 04/10/2018].

[4] R. L. Cook and K. E. Torrance: A reflectance model for computer graphics. SIGGRAPH Comput. Graph., 15:307–316, August 1981.

[5] Blinn, J.F. Models of light reflection for computer synthesized pictures. In Proceedings of SIGGRAPH '77

[6] McGuire, M., Osman, B., Bukowski, M., Hennessy, P.: The alchemy screen-space ambient obscurance algorithm. In: HighPerformance Graphics 2011 (2011)

[7] MITTRING, M. 2007. Finding next gen: Cryengine 2. In ACM SIGGRAPH 2007 courses.

[8] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, "Real-Time Rendering - Third Edition", 2008

[9] Louis Bavoil, Miguel Sainz: Screen Space Ambient Occlusion, 2008

[10] BAUER F., KNUTH M., BENDER J.: Screen-space ambient occlusion using A-buffer techniques. In Computer-Aided Design and Computer Graphics (Nov. 2013), IEEE.

[11] Kurt Akeley, Pat Hanrahan: Real-Time Graphics Architecture, 2001 http://www.graphics.stanford.edu/courses/cs448a-01-fall

[12] Louis Bavoil, Miguel Sainz, Rouslan Dimitrov: Image-space Horizonbased Ambient Occlusion. In: ACM SIGGRAPH 2008 Talks. SIGGRAPH '08. Los Angeles, California: ACM, 2008

[13] K. Perlin and E. M. Hoffert: Hypertexture. SIGGRAPH Comput. Graph., 23(3):253–262, July 1989.

[14] RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating dynamic global illumination in image space. In I3D '09

[15] SEGOVIA, B., IEHL, J.-C., MITANCHEY, R., AND PEROCHE, B.2006: Non-interleaved Deferred Shading of Interleaved Sample Patterns. In SIGGRAPH/Eurographics Graphics Hardware.

[16] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," Sixth International Conference on Computer Vision, New Delhi, 1998.

[17] EVERITT, C. 2001. Interactive order-independent transparency.

[18] John Chapman. SSAO Tutorial, 2011. http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html [Retrieved 01/08/2018]

[19] A Fast and Stable Feature-Aware Motion Blur Filter Jean-Philippe Guertin, Morgan McGuire, and Derek Nowrouzezahrai NVIDIA Technical Report NVR-2013-003, November 26, 2013.

[20] Cale Hunt: What is asynchronous reprojection and how do I use it? , April 2017. https://www.vrheads.com/what-asynchronous-reprojection-and-how-do-i-use-it. [Retrieved 06/10/2018]