Real-time ambient lighting with raytracing

András Fridvalszky[1] and Balázs Tóth[1]

Budapest University of Technology and Economics

Abstract. Ambient occlusion is a popular lighting effect in modern rendering engines. One of its implementations is Screen Space Ambient Occlusion (SSAO), which is generally used in real-time applications. It is a relatively cheap effect which can substantially improve the realism of a scene. It also has its drawbacks, such that it is unable to take into account the effect of invisible geometries and can produce unnatural artifacts. These could be solved by using ray tracing based techniques, but previously they were too slow for real-time purposes. Now recent advancements in this area made it possible to apply methods like these without compromising performance by using hardware acceleration for ray tracing. In this paper, we review the important aspects of hardware accelerated ray tracing and ambient occlusion. We also propose an extension for the ray tracing based implementation that can capture thin objects like drapes with a handful of samples.

1 Introduction

Ray tracing as a technique was first described in [1], but for real-time applications like computer games, it could not be used effectively until recently. The hardware was optimized for the rasterization based pipelines and could not handle a large number of ray-triangle intersection calculations required to create high-quality results. Still, ray tracing based algorithms were researched and developed too because they were used for offline rendering purposes, like movies, where the execution speed was less critical.

Compared to rasterization ray tracing provides many benefits. Common effects like reflections and shadows can be achieved naturally, with better quality and fewer artifacts. With rasterization clever techniques and optimizations were required to maintain real-time frame rates. In 2018 NVIDIA introduced a new line of GPUs with hardware based ray tracing support. Using specialized RT cores the GPU could process ray-triangle intersections much faster than before, and so ray tracing became usable for computer games too. Many popular game engines implemented hybrid renderers where rasterization based pipelines were supplemented with ray tracing based effects to provide more realism. Until recently, only owners of an NVIDIA RTX GPU could benefit from these, but in 2020 AMD also introduced new GPUs with support for ray tracing. The extensions to use these new features were also standardized and added to the Vulkan API, so developing for multiple platforms is much easier.

While fully replacing rasterization with ray tracing could become possible in the future, for now it is only used for certain effects which are easier to compute and provide better realism. Dynamic ambient occlusion is one such technique. In rasterization based pipelines screen space techniques were used, but they can not take into account invisible parts of the scene. Now with ray tracing, these effects can be further refined to create artifact free and immersive lighting.

In this paper, we make an overview of the ray tracing pipeline, its features and how they can be used to implement ambient occlusion and an approximation of global illumination.

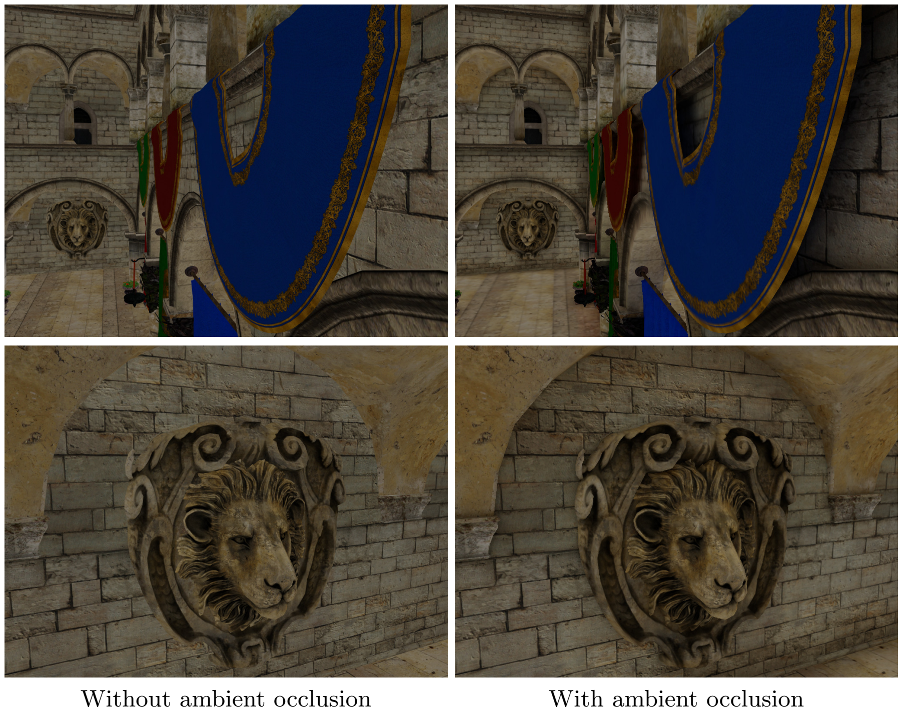

Fig. 1. Results of applying ambient occlusion in the Sponza scene. By using a transfer function color bleeding can be achieved, demonstrated with bluish tint on the wall behind the curtain (top right).

2 Related work

Nearby occlusions, accessibility, or openness of points on a surface were first examined with the objective of ambient lighting calculations in [2, 3], where the *obscurances method* was introduced. The obscurances method, which is also called the *ambient occlusion* [4, 5] computes just how "open" the scene is in the neighborhood of a point and scales the ambient light accordingly. Obtaining their radiance, occluders can be treated as indirect light sources, so even color bleeding effects can be simulated [6–9].

Until recently, a physically correct approach would be too expensive computationally when dynamic scenes need to be rendered in real-time, therefore the mentioned models simplify the problem. For a better approximation of the ambient lighting, Bavoil et. al [10] proposed a hybrid technique to combine the classic screen space approach with the new hardware accelerated ray tracing to improve the robustness while maintaining a low sample count. Gautron et. al [11] proposed a full ray tracing based technique where the sampling noise is filtered in world space with various techniques to reach real-time frame rates with complex scenes.

Another promising direction could be to use neural networks to predict the ambient occlusion. Zhang et. al [12] recently proposed one such technique with a detailed comparison to existing methods.

3 Overview of hardware accelerated ray tracing

Hardware accelerated ray tracing provides new possibilities for the developer, who in turn can create a more impressive virtual world for the end-user. But to achieve a pleasing result and to maintain acceptable frame rates, especially on the lower end of the GPUs, we must have a good understanding of the underlying implementation. Just as with other parts of the Vulkan API, we are presented with multiple ways of doing the same thing. To make the right choice, we need to know the advantages and the disadvantages, the intended use cases of the different features. In following sections we will make an overview of how the ray tracing API works, and highlight some of the more important choices, the developer must make. We will use terminology based on the Vulkan API, but both the OptiX and the Microsoft DXR API are very similar.

3.1 Acceleration structure

The most performance demanding part of ray tracing is to solve the visibility problem. For every ray, we need to determine which triangle or object it intersects. We need to create an acceleration structure so we can determine these efficiently. However, we must also take into account that scenes in modern games are highly dynamic. We will have to update it in some ways before rendering every frame. The inner workings of this structure are managed by the implementations, but we are still left with a lot of flexibility to tune it for our own usage.

For the developers, two parts of the acceleration structure are visible. The bottom level acceleration structure (BLAS) and the top level acceleration structure (TLAS). A BLAS represents an object or a group of objects that are close together, while the TLAS is a collection of BLAS-s. We can trace rays against a TLAS, so in simple cases we can think about it as a representation of our scene on the GPU. A BLAS can hold objects that are built up by triangles or objects that are represented by an axis-aligned bounding box. In the first case, handling of the ray-triangle intersections is implemented by the hardware. In the second case we must provide an intersection shader, which will be called by the implementation when required. We will discuss this in detail later on. The TLAS refers to already created BLAS-es. It can refer to the same BLAS multiple times with different transformations (like instancing).

When building any one of the two types of the acceleration structure we can decide if we would like to optimize it for fast ray tracing or for fast building. We can also decide if we would like to update it on the fly (which will be potentially faster) or just rebuild it from scratch. For example, an object without animations should be built for fast ray tracing because its inner structure will not be modified (transformations can be handled by the TLAS). On the other hand, an object with animations is much more difficult to handle. The BLAS must be updated every frame, for example using a compute shader to calculate the animated positions of the vertices. But if the animation contains frames where there are large differences in positions, then an updated structure will not be efficient for ray queries. In this case, a rebuild can become necessary. It is also possible to pre-build multiple acceleration structures for the same object in different poses and update the one that is closest to the actual pose. These decisions should be made on a case-by-case basis, and profiling should be used where it isn't clear which one is the better.

If possible, multiple objects should be grouped together inside a BLAS. It's a good idea to minimize the empty space inside its axis-aligned bounding box. The frequency of updates should also be kept in mind. It would not be a good idea to group static and animated objects together, just because they are close to each other.

For the TLAS fast tracing should be preferred. It can also be fully rebuilt every frame for dynamic scenes to not lose its efficiency. Culling can also be applied to remove objects that are not interfering with the current view. Of course, we have to keep in mind that invisible objects can still affect the scene, for example, through shadows and reflections.

3.2 Shaders and pipeline

With rasterization, we are rendering one object at a time, and so we can bind the actual shaders, uniform variables and vertex buffers one by one. (Of course, grouping them together is a good idea.) On the other hand, with ray tracing, a ray can intersect any object currently present in the scene, so every data must be available on the GPU. This means all the textures, vertex buffers and also the shaders used by the different materials. They must be linked to the objects, encapsulated in BLAS-es and referred by the TLAS. To solve this problem, multiple new concepts were introduced.

The first one we shall discuss is the hit group. It represents a collection of shaders to call when a ray intersects an object. A hit group is directly linked to an object inside a BLAS. It can contain multiple shaders with different purposes. If the object is described by a bounding box, it must contain an intersection shader (otherwise, it must not). The implementation will call it when a ray intersects the bounding box, and it must decide if a true intersection happened. It can report multiple intersections as well. One usage could be to render an implicit surface described by a mathematical function. The closest hit shader will be called for the intersection that is closest to the origin of the ray. The any hit shader will be called for all intersections (including the closest).

Ray queries can be started in the ray generation shader. When a ray tracing operation is requested, a certain number of ray generation shader instances are spawned (just like compute shaders). These can create a ray and query a TLAS for intersections. A hit is determined as discussed above, and the relevant shaders are executed. Data can be both sent from and returned to the ray generation shader. If no intersections are found for a ray, then the active miss shader is called. This can be selected when a ray query is started.

During execution, a closest or an any hit shader can create another ray query recursively. Deep recursions should be avoided. One reason is that implementations are not required to support recursion at all. It should be replaced by cycles in the ray generation shader. For example, the closest hit shader can return the surface attributes, and based on this, the ray generation shader can create another query to calculate reflections. Deep recursive pipelines can have an impact on the performance, but one level deep recursion (shadow rays for example) is acceptable.

The last type of shader usable with ray tracing is the callable shader. A pipeline can contain multiple different callable shaders. They are identified by an index and can be called from any shader stage. They are not needed to be linked against the caller, and they can be selected based on dynamic values. For example, to do the shading the closest hit shader could call a different callable shader based on the type of the material.

The ray tracing pipeline is basically the collection of these shaders. What left is how to decide which one to call. To solve this, we will need the so called shader binding table. It is an array containing the handlers to all the shaders. Identification is done by calculating the index of the correct handler. For the ray generation shader, this must be done on the CPU side during command recording. The miss shader is identified by its index during the creation of a ray query. Above we already discussed the callable shader, so only the hit group selection remains.

When a BLAS reference is added to the TLAS, besides the transformation matrix, the base address for the hit groups can be selected too. As we discussed earlier, the BLAS can contain multiple geometries and they can use different shaders. The shader binding table must contain all hit groups for the geometries in order, starting at the base offset given for the BLAS reference. When an intersection is found the correct index will be computed by the GPU. Sometimes, based on what is the purpose of the ray, certain optimizations can be done. For example, occlusion queries could need less work than a reflection query. To do this, multiple ray types can be specified and different shaders can be stored in the shader binding table. For each geometry the table should contain a handler, one after the other for each type of ray.

4 Ambient occlusion

For the sake of simplicity, ambient occlusion or obscurances models assume that the surfaces are diffuse and have albedo $a(\boldsymbol{s})$ and consequently their BRDF is $f_r(\boldsymbol{s}) = a(\boldsymbol{s})/\pi$. According to the rendering equation, the reflected radiance L^r in *shaded point* $\boldsymbol{s}$ can be obtained as:

$$L^r(\boldsymbol{s}) = \int\limits_{\Omega} L^{in}(\boldsymbol{s}, \boldsymbol{\omega}) \frac{a(\boldsymbol{s})}{\pi} \cos^+ \theta \mathrm{d}\omega, \tag{1}$$

where Ω is the directional sphere, $L^{in}(\boldsymbol{s}, \boldsymbol{\omega})$ is the incident radiance from direction $\boldsymbol{\omega}$ and θ is the angle between illumination direction $\boldsymbol{\omega}$ and the surface normal.

If no surface is seen from $\boldsymbol{s}$ at direction $\boldsymbol{\omega}$, then shaded point $\boldsymbol{s}$ is said to be *open* in this direction, and incident radiance L^{in} is equal to ambient radiance L^a. If there is an occluder nearby, then the point is called *closed* at this direction and the incident radiance is the radiance of the *occluder point* $\boldsymbol{o}$.

Note that this means that the radiance of arbitrary far occluder points may have an effect on the shaded point. However, this does not meet our intuition and everyday experience that the effect of very far surfaces becomes negligible. This experience is due to that the real space is not empty but is filled by participating media. If the extinction coefficient is σ_t and the albedo is 1 in the participating media, then the radiance along a ray changes as:

$$L^{in}(\boldsymbol{s}, \boldsymbol{\omega}) = \exp(-\sigma_t d) L^r(\boldsymbol{o}) + (1 - \exp(-\sigma_t d)) L^a$$

where d is the distance between the shaded point and the occluder point. Note that in this equation factor

$$\mu(d) = (1 - \exp(-\sigma_t d)) \tag{2}$$

and its complement $1 - \mu(d) = \exp(-\sigma_t d)$ express the effects of the ambient lighting and of the occluder on the shaded point, respectively. The effect of the occluder diminishes with the distance. Function μ can be interpreted as a fuzzy measure that defines how strongly the given direction belongs to the set of open directions based on distance d of the occlusion at this direction.

The exponential function derived from the physical analogy of participating media has a significant drawback [3]. As it is non-zero for arbitrarily large distances, very distant surfaces need to be considered that otherwise have a negligible effect. Thus, for practical fuzzy measures, we use functions that are non-negative, monotonically increasing from zero and reach 1 at finite distance R instead of only converging to 1 when the distance goes to infinity. This allows the consideration of only those occlusions that are nearby, i.e. in the neighborhood sphere of radius R. The particular value of R can be set by the application developer. When we increase this value, shadows due to ambient occlusions get larger and softer.

To define the fuzzy measure that increases from zero to one in $[0, R]$, we can use a simple polynomial:

$$\mu(d) = \left(\frac{d}{R}\right)^{\alpha}.$$

Using the fuzzy measure, the reflected radiance of the shaded point can be expressed in the following way:

$$L^r(\boldsymbol{s}) = a(\boldsymbol{s})\,(L^a O(\boldsymbol{s}) + I(\boldsymbol{s}))$$

where $O(\boldsymbol{s})$ is the ambient occlusion (obscurance) of the shaded point [2, 3]:

$$O(\boldsymbol{s}) = \frac{1}{\pi}\int\limits_{\Omega} \mu(d)\cos^{+}\theta \mathrm{d}\omega, \tag{3}$$

and $I(\boldsymbol{s})$ is the irradiance due to nearby indirect lighting:

$$I(\boldsymbol{x}) = \frac{1}{\pi}\int\limits_{\Omega} (1-\mu(d))L^r(\boldsymbol{o})\cos^{+}\theta \mathrm{d}\omega. \tag{4}$$

This integral is traced back to the ambient occlusion. Replacing occluder radiance $L^r(\boldsymbol{o})$ by the average of surface radiance values in the neighborhood of the shaded point $\tilde{L}^r(\boldsymbol{s})$, we can express the irradiance as:

$$I(\boldsymbol{s}) \approx \frac{1}{\pi}\int_{\Omega}[1-\mu(d)]\tilde{L}^r(\boldsymbol{s})\cos^{+}\theta d\omega = \tilde{L}^r(\boldsymbol{s})(1-O(\boldsymbol{s})).$$

The original definition of ambient occlusion [4, 5, 13] can be interpreted as the simplification of the obscurance when the fuzzy measure becomes a clear separation of occlusion closer than R and no occlusion closer than R. On the other hand, it was also proposed to approximate the solid angle and the average direction where the neighborhood is open, thus we can use environment map illumination instead of the ambient light. However, nowadays many authors include a weighting function into the ambient occlusion formula, thus ambient occlusion has become equivalent to the original obscurance model. As the term ambient occlusion has got more popular than the older obscurance, we shall name the general method as ambient occlusion.

The solution of equation 4 requires some estimation of the radiance $L^r(\boldsymbol{o})$ reflected off the occluding points, which is responsible for indirect lighting. The first approach to extend the obscurances model with local indirect lighting was the *spectral obscurances method*, which took average spectral reflectivities of the neighborhood and the whole scene into account, thus even color bleeding effects could be cheaply simulated [14]. Bunnel [15] extended ambient occlusion as well to incorporate color bleeding.

In our method we used the following formula of local indirect lighting (equation 4), which can be traced back to the computation of ambient occlusion. Replacing the occluder radiance $L^r(\boldsymbol{o})$ by average $\tilde{L}^r(\boldsymbol{o})$ of surface radiance values in the neighborhood of the shaded point, we can express the indirect lighting as:

$$I(\boldsymbol{s}) = \frac{1}{\pi} \int\limits_{\Omega} (1 - \mu(d)) L^r(\boldsymbol{o}) \cos^+ \theta \mathrm{d}\omega \approx \tilde{L}^r(\boldsymbol{s})(1 - O(\boldsymbol{s})).$$

The average surface radiance can be obtained during tracing the occlusion ray. If the closest hit point along the ray is closer than a predetermined limit the diffuse component of the surface material is used as an approximation.

5 Implementation of ambient occlusion and indirect lighting

The discussed algorithm is implemented as a two render pass. The first pass calculates the direct lighting, the ambient occlusion factor and the ambient radiance for every visible surface point. To maintain real-time performance we approximate the occlusion factor with a small number of occlusion rays. To generate the directions of the occlusion rays we use an importance sampling-based approach. As we cannot store a handful of sample directions for every surface point of the scene we have to decide whether we use the same sample set for every point or use some perturbation. The same samples would make the error correlated and results in "stripes", applying perturbation would cause dot noise. Unfortunately, both of these types of noise are quite disturbing.

In order to reduce the error without taking an excessive number of samples, we apply interleaved sampling [16]. Interleaved sampling takes advantage of the estimates in neighboring pixels, as it uses different sets of samples of the pixels of a 4×4 pixel pattern in the first pass. The errors in the pixels of a 4×4 pixel pattern are uncorrelated, which can be successfully reduced by a low-pass filter of the same size in the second rendering pass. When implementing the low-pass filter, we can also check whether the depth difference between the current and neighboring pixels exceeds a given limit. If it does, we do not include the neighboring pixels in the averaging operation.

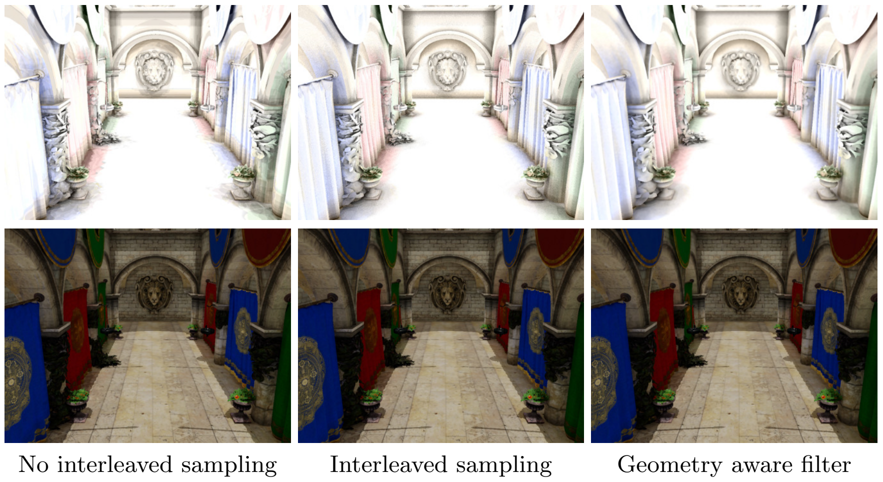

Fig. 2. Without interleaved sampling visible artifacts can be seen on the resulting image. By using different samples the noise becomes less prominent for the observer. Finally the geometry aware low-pass filter substantially improves the results.

5.1 Improving sampling accuracy for thin objects

Even ray traced ambient occlusion can cause certain artifacts to appear. When a thin object is near a surface, parallel to its normal, as seen on figure 3, very few, possibly none of the sampled rays will hit the object. This results in an artifact, similar to light bleeding in standard shadow mapping. Rays originating from the neighboring surface points will have a much higher probability to detect the occluding object. The resulting effect is the incorrect lightening of an inner area directly under the object compared to the surrounding surface. This could also happen with screen spaced techniques, where the limited nature of the available data makes it very hard to remove the artifact.

With ray tracing we can try to detect these thin geometries with another set of rays. We will use the ones used for the sampling of local ambient lighting, which did not catch a geometry to generate this next batch of rays that will have a high probability to find the potential thin occluder. We failed to detect the object because it is near parallel to our first batch of rays and thin enough to fit inside the volume between them. If it is truly an occluder, then it must have a large enough volume alongside the direction of the rays. According to this, we collect the ones that did not catch geometry and connect every endpoint to each other with new sample rays. These will be able to hit the thin geometry through its larger surface. The idea is demonstrated on figure 4.

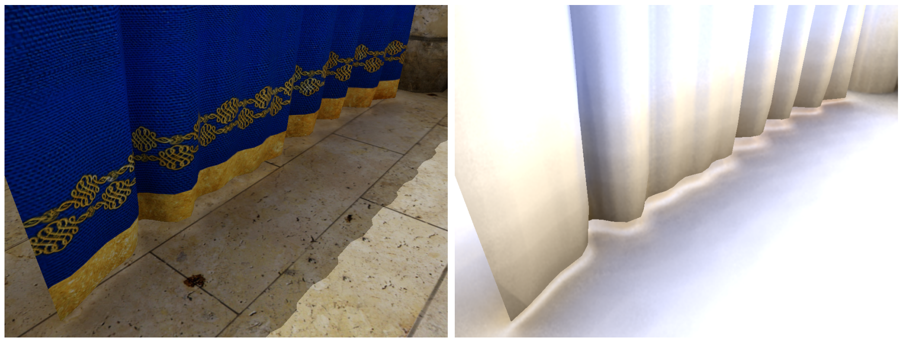

Fig. 3. The area directly under the curtain incorrectly receives more ambient light than its surroundings.

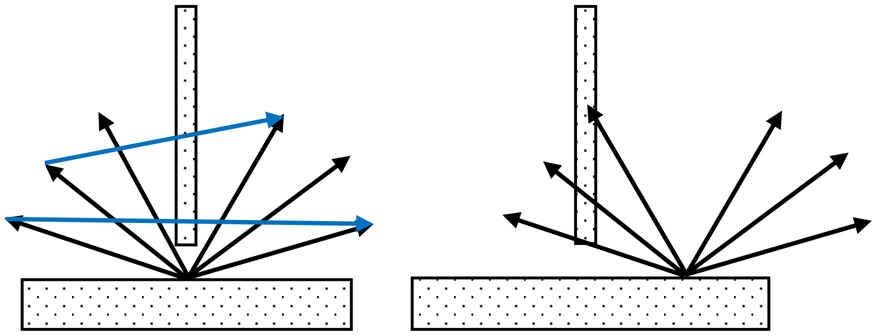

Fig. 4. Schematic figure to demonstrate the idea of detecting thin objects. Rays originating from a point directly under the occluder (left) has less chance to detect it compared to ones that are further away (right). However the rays shown in blue can successfully catch the previously unseen geometry.

We take each new ray into account with a weight proportional to its length. This corresponds to the size of the sampled volume. We take the minimum of the resulting average and the result of the first batch of rays. This way, we fill in the potential holes left by the standard method as seen on figure 5.

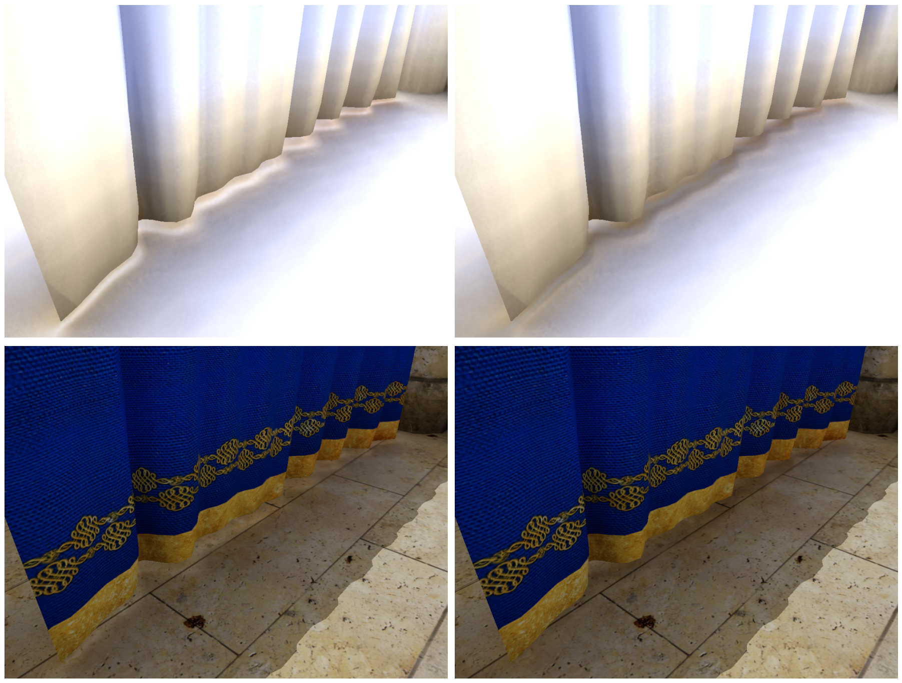

Fig. 5. Demonstration of improvements over the standard technique (left) with the additional rays (right). Also note that the side of the pillar, close to the curtain is correctly darkened in the improved version.

6 Results

The quality of the results heavily depends on the rays used to sample ambient lighting. Increasing the number of rays obviously improves quality, given the nature of the algorithm but requires more time to compute. By choosing appropriate sample rays, providing a good coverage of the hemisphere over the surface, using the above mentioned technique to detect thin occluders and using interleaved sampling we can reduce the number of samples per pixel, substantially improving performance. The quality of the ambient occlusion map becomes worse but the final results are barely affected.

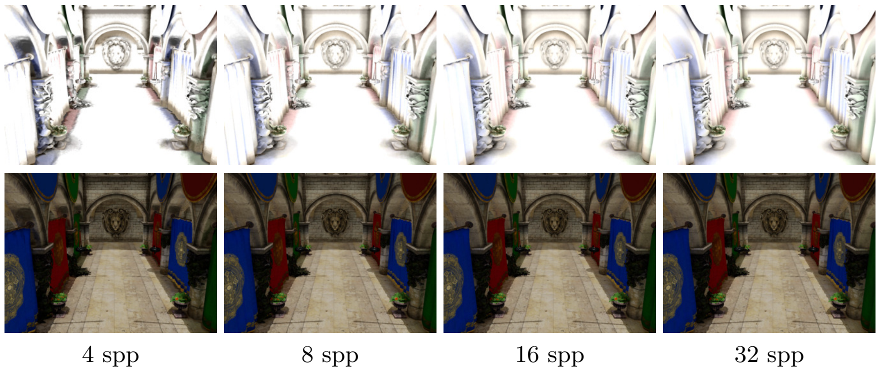

Fig. 6. Demonstration of image quality given the number of sampels per pixel.

Based on our experiments real-time frame are rates achieved by using up to 8 samples and more than that does not visibly improve the image quality. Because of the transfer function using only 4 samples, it results in visible noise. Our thin object detection for 8 samples only increases frame time by about 2 ms on Full HD image with 1920 × 1080 pixels.

Acknowledgements

The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program.

References

1. Turner Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, pages 4–es. 2005.
2. S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. In *Proceedings of the Eurographics Rendering Workshop*, pages 45–56, 1998.
3. A. Iones, A. Krupkin, M. Sbert, and S. Zhukov. Fast realistic lighting for video games. *IEEE Computer Graphics and Applications*, 23(3):54–64, 2003.
4. Hayden Landis. Production-ready global illumination. Technical report, SIGGRAPH Course notes 16, 2002. http://www.renderman.org/RMR/Books/sig02.course16.pdf.gz.
5. M. Pharr and S. Green. Ambient occlusion. In *GPU Gems*, pages 279–292. Addison-Wesley, 2004.
6. A. Méndez, Mateu Sbert, and Jordi Catá. Real-time obscurances with color bleeding. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 171–176, New York, NY, USA, 2003. ACM.
7. Tamás Umenhoffer, Balázs Tóth, and László Szirmay-Kalos. Efficient methods for ambient lighting. In *Proceedings of the 25th Spring Conference on Computer Graphics*, pages 87–94, 2009.

8. László Szirmay-Kalos, Tamás Umenhoffer, Balázs Tóth, László Szécsi, and Mateu Sbert. Volumetric ambient occlusion for real-time rendering and games. *IEEE Computer Graphics and Applications*, 30(1):70–79, 2009.
9. Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, 2009.
10. Louis Bavoil, Edward Liu, Peter Shirley, and Morgan McGuire. Hybrid ray-traced ambient occlusion, August 2018. Poster at HPG18 ACM SIGGRAPH / Eurographics High Performance Graphics.
11. Pascal Gautron. Real-time ray-traced ambient occlusion of complex scenes using spatial hashing. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks*, pages 1–2, 2020.
12. Dongjiu Zhang, Chuhua Xian, Guoliang Luo, Yunhui Xiong, and Chu Han. Deepao: Efficient screen space ambient occlusion generation via deep network. *IEEE Access*, 8:64434–64441, 2020.
13. J. Kontkanen and T. Aila. Ambient occlusion for animated characters. In *Proceedings of the 2006 Eurographics Symposium on Rendering*, 2006.
14. A. Mendez, M. Sbert, J. Cata, N. Sunyer, and S. Funtane. Real-time obscurances with color bleeding. In Wolfgang Engel, editor, *ShaderX 4: Advanced Rendering Techniques*. Charles River Media, 2005.
15. M. Bunnel. Dynamic ambient occlusion and indirect lighting. In M. Parr, editor, *GPU Gems 2*, pages 223–233. Addison-Wesley, 2005.
16. A. Keller and W. Heidrich. Interleaved sampling. In *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, pages 269–276, 2001.