Eurographics Symposium on Rendering - Experimental Ideas & Implementations (2017)
P. Sander and M. Zwicker (Editors)

# VAO++: Practical Volumetric Ambient Occlusion for Games

J. Bokšanský[1], A. Pospíšil[1] and J. Bittner[2]

[1]Project Wilberforce, Czech Republic
[2]Czech Technical University in Prague, Czech Republic

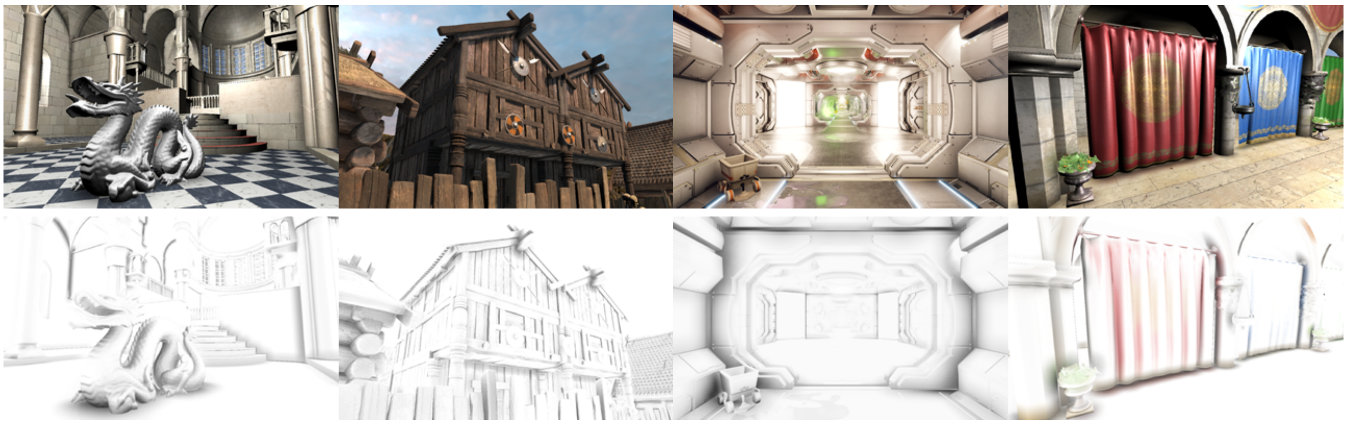

**Figure 1:** *Snapshot of the test scenes rendered in Unity game engine with VAO++ applied (top row) and VAO++ component only (bottom row). The rightmost scene shows VAO++ with color bleeding enabled.*

**Abstract**

*Ambient occlusion is one of the commonly used methods to increase visual fidelity in real-time rendering applications. We propose several extensions of the recently introduced volumetric ambient occlusion method. These extensions improve the properties of the methods with a particular focus on the quality vs performance tradeoff and wide applicability in contemporary games. We describe the implementation of the proposed algorithm and its extensions. We implemented the method as a camera effect within the Unity game engine. The results show that our implementation compares favorably with the standard ambient occlusion in Unity both in terms of quality and speed.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

## 1. Introduction

The contemporary graphics hardware is powerful enough to render complex dynamic scenes with detailed geometry, realistic materials, and various illumination effects in real-time. However, physically based simulation of light transport is still far too complex for real-time applications such as games. To counteract this issue numerous techniques approximating different illumination effects have been designed, such as shadow mapping to handle shadows, reflection maps to handle specular reflections, or light maps for precomputing global illumination. One of the popular techniques is ambient occlusion [ZIK98,IKSZ03,Lan02] that aims to capture the occlusion due to nearby geometry and thus to compute its influence of ambient lighting. In many scenes, ambient lighting is an important illumination component, and consequently ambient occlusion greatly contributes to the perception of realism of the rendered images.

The first ambient occlusion methods worked in object space and evaluated the ambient occlusion in a preprocess [ZIK98, Lan02]. With the introduction of screen space ambient occlusion [Mit07, SA07] it was possible to approximate ambient occlusion in real-time using just the information stored in the depth buffer. Several powerful screen space ambient occlusion methods have been proposed using a different interpretation of the depth-buffer information and the associated ambient occlusion evaluation. One of the more recent techniques is the volumetric ambient occlusion (VAO) proposed by Szirmay-Kalos et al. [SKUT*10]. This method formulates the problem as a volumetric integral providing robust results even with a relatively low number of samples.

We propose the VAO++ method that consists of several practical extensions of VAO that aim to maximize the performance for real-time applications such as games. In particular, we propose adaptive sampling and culling pre-pass that speed up the algorithm with minimal influence on the quality of the output. Our solution achieves up to 2.4x speedup with respect to the original VAO method and 1.3x speedup with respect to the built-in Unity SSAO implementation. We also describe several other practical details that increase the perceived quality of the results such as luminance sensitivity, depth range check, and efficient interleaved sampling with a compact sample set.

We describe the implementation of VAO++ in the Unity game engine. The results indicate that the method competes favorably with the standard ambient occlusion implemented in Unity both concerning speed and perceived quality.

## 2. Related work

One of the first attempts to handle ambient lighting is the ambient term in the classical Phong illumination model [Pho75]. This term accounts for global ambient illumination, but it does not handle occlusion due to nearby geometry. Zhukov et al. [ZIK98] and Iones et al. [IKSZ03] were the first to observe that the modulation of incoming light could be precomputed by evaluating the *accessibility* of the points on the surfaces with respect to incoming ambient light. They used a continuous function of distance to modulate the contribution of nearby geometry to the *obscurance* or *ambient occlusion* of a point. The ambient occlusion has been quickly adopted by the industry [Lan02]. In many cases a simplified version of ambient occlusion has been used which did not take the distance into account and evaluated the unoccluded fraction of the hemisphere within a given distance threshold.

To support dynamic scenes Kontkanen et al. [KL05] proposed ambient occlusion fields and later extended to method to animated characters [KA06]. Bunnnell [BE05] proposed a method for handling scenes with deforming surfaces. Mendez et al. [MSC03] proposed to account for color bleeding effects. Further extensions of these methods have been proposed by Hoberock and Jia [HJ08] and Christensen [Chr08].

The previously mentioned methods rely on ray tracing or various surface discretizations. A new path towards handling complex fully dynamic scene at interactive rates appeared with the introduction of screen space ambient occlusion methods [HJ08, Mit07, SA07, Ngu07, CAM08]. These methods evaluate the ambient occlusion in real-time based on the content of the depth buffer. The screen space methods were extended to also handle directional occlusion [RGS09] for more accurate environmental lighting and color bleeding. Sainz et al. [Sai08] proposed the horizon based ambient occlusion which evaluates the ambient occlusion by reconstructing the horizon as seen from the shaded point. Szirmay et al. [SKUT*10] proposed a volumetric ambient occlusion that uses a clever reformulation of the screen space AO evaluation to volumetric integration. We use this method as a basis for our optimizations and discuss some practical issues of integrating the method into a contemporary game engine.

The paper is organized as follows: In Section 3 we outline the VAO algorithm. Section 4 presents the extensions of the VAO++ algorithm. Section 5 describes implementation in Unity. Section 6 presents the results and finally Section 7 concludes the paper.

## 3. VAO Algorithm Outline

In this section we provide a brief outline of the VAO algorithm [SKUT*10]. Much like other screen-space solutions VAO approximates occlusion by considering local occluders only. Most screen space ambient occlusion method analyze the contents of the depth buffer lying inside a hemisphere of a given radius ($R$). The volume of the hemisphere is sampled and the ambient occlusion is computed as the fraction of unoccluded samples. Contrary to this, VAO uses a clever transformation of the sampling domain to a smaller *tangent sphere* of radius $R/2$ moved along the surface normal (Figure 2). The ambient occlusion is computed as a fraction of unoccluded volume of the tangent sphere.

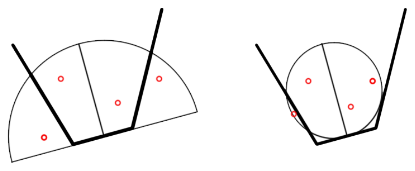

**Figure 2:** *The comparison of searched volumes of conventional AO (hemisphere) and VAO (tangent sphere).*

Such transformation creates a tighter local neighbourhood (the tangent sphere has fourth of the volume of the usually used hemisphere) leading to a more efficiently sampled volume. It also simplifies implementation: there is no need to calculate cosines and the generated samples all lie within the disk of radius $R/2$ located in the center of tangent sphere perpendicular to Z-Axis (in camera space).

Each sample represents a volume (pipe) intersecting the tangent sphere that corresponds to a ray from the camera to the given sample. For each sample an unoccluded part of the corresponding pipe is calculated (the total length of unoccluded pipe can be precalculated for each sample as an optimization). Then for each sample one of the following three cases occurs (Figure 3):

- Case A: Occlusion happens inside the tangent sphere
- Case B: Either no occlusion or occlusion happens deeper in the scene
- Case C: Occlusion occured before reaching the tangent sphere.

The volumes of the unoccluded pipe segments are summed and represent an open, unoccluded volume which is divided by the total volume of the tangent sphere. The volumetric ambient occlusion

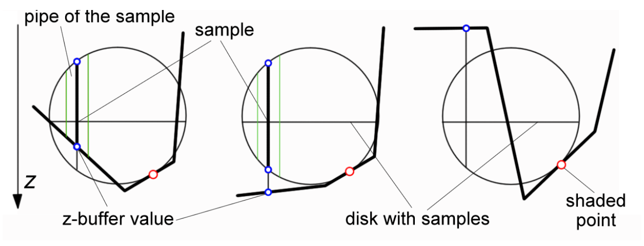

**Figure 3:** *Three possible cases for line samples. Case A: the pipe is partially occluded, Case B: the pipe is completely unoccluded and Case C: the pipe is completely occluded.*

$V(\vec{x})$ of point $x$ is computed by uniformly sampling the disc spanning the tanget sphere as:

$$V(\vec{x}) = \frac{\int_S \mathcal{I}(\vec{p})dp}{|S|} \approx \frac{3}{2Rn}\sum_{i=1}^{n}\Delta z_i = \frac{1}{F}\sum_{i=1}^{n}\Delta z_i, \tag{1}$$

where $n$ is the number of samples, $\Delta z_i$ is the length of the pipe corresponding to $i-th$ sample, $R$ is the width of the AO kernel, and $F$ is the volume of the tangent sphere $|S| = 4(R/2)^3\pi/3$ divided by $R^2\pi/(4n)$. Analytical computation of $F$ is possible but we use a numerical approximation as it produces better results due to correlated error of the estimators [SKUT*10]:

$$F \approx R\sum_{i=1}^{n}\sqrt{1-x_i^2-y_i^2} \tag{2}$$

The VAO algorithm runs in three passes: (1) calculate occlusion into a texture, (2) apply low pass filter removing high frequency noise, (3) perform shading, i.e. apply the occlusion to the color buffer.

## 4. The VAO++ Algorithm

We extend original VAO algorithm to improve its performance and to make it more efficient in common application use cases. The extensions and other important parts of the proposed VAO++ algorithm are described in this section.

Our extensions work effectively for wide range of scenes. The heuristics they depend on are focused on gaining performance in areas where the difference is less noticeable – distant geometry in case of adaptive sampling and open surfaces for culling pre-pass. These methods are not general but work well especially in games where such cases commonly appear.

### 4.1. Adaptive Sampling

The existing AO methods usually use a globally specified quality setting including the number of samples to use. The idea of *adaptive sampling* extension is to use fewer samples further from camera where the sampling radius is smaller (due to perspective projection) and therefore lower number of samples should be sufficient to achieve the same level of quality. In other words we try to maintain the same sampling density in post-perspective image space.

In VAO the sampling kernel always displays as a disc on the screen (not a hemisphere as in other algorithms) and thus it is easy to select the appropriate number of samples based on the surface area of the disc after perspective projection.

We let the user set the diameter of the disc in relative screen space units for which the lowest number of samples should be used. Because of this the method works the same way across different resolutions and it also enables us to give user multiple pre-defined quality options. The number of samples used on this pre-defined distance is set to the minimum (i.e. two samples) and we double the number of samples every time surface area of the sampling disc doubles until the maximum number of samples is reached.

The implementation selects from the precomputed sample sets with 2, 4, 8, 16 or 32 samples. We send all these samples to the GPU shader as a single array uniform of size 62. Such sample set fits the practical limit of the array size sent to GPU (100 items) and thus supports as many target platforms as possible.

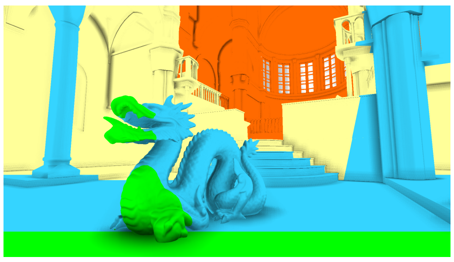

**Figure 4:** *Visualization of the levels of adaptive sampling – back to front: 4, 8, 16, 32 samples.*

### 4.2. Culling Pre-pass

In most scenes, the ambient occlusion appears only in some areas of the image (corners, cracks etc.). In other areas, the calculation is unnecessary and could be omitted. The culling pre-pass is a technique designed to speed-up AO calculation by omitting it in areas that are likely to have no occlusion. We first calculate AO in a lower resolution texture (we use $1/8$ resolution of the final image) to estimate areas where AO occurs. We also always use 8 samples for this pass. Calculating AO in this low resolution texture is theoretically 64-times faster and therefore very cheap. In the second pass we only calculate AO for areas which were occluded in the low resolution texture. For higher resolutions and sampling kernel sizes, this step significantly improves the performance with minimal quality loss. It was designed with 'single pass stereo rendering' technique for virtual reality in mind, where the effect is applied to a relatively large texture.

Checking for estimated occlusion in the second pass is done via four texel fetches from downsized AO texture in given pixel corners. Texture interpolation mode is set to linear so we get an interpolated result from eight surrounding pixels covering $24 \times 24$ area. When at least one of these pixels exhibits occlusion, we calculate full AO – otherwise the pixel is a candidate for culling (Figures 5 and 6).

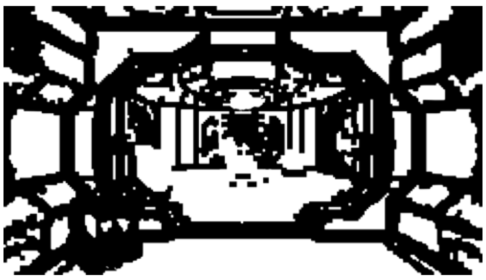

**Figure 5:** *Illustration of culling pre-pass. Areas where no occlusion is estimated are culled (shown in white). Areas that will be calculated in full detail are shown in black.*

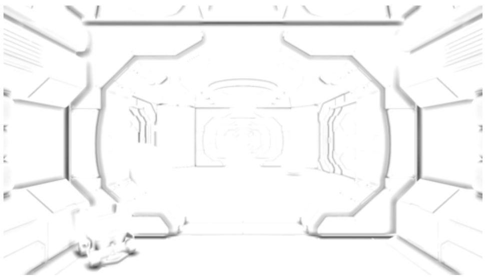

**Figure 6:** *Resulting scene with culling pre-pass enabled.*

For handling the culling candidate pixels we settled with two variants of this technique. First we call 'greedy' and it completely omits the calculation where no AO is estimated. This can possibly lead to flickering white lines or squares of size $8 \times 8$ when estimation fails to detect small details. When using large AO radius, greedy variant can produce little to no artifacts and can be used for higher performance boost. This is obviously scene dependent and should it become an issue, we fall back on to a second 'careful' variant of the method. This technique calculates occlusion with 4 samples in areas where no AO is estimated regardless of selected quality. Four samples are enough to provide adequate quality, because these areas only contain small details (less than $8 \times 8$ pixels in size at most).

The quality loss of careful culling pre-pass is almost imperceptible and is turned on as a default setting in our implementation (Figure 7).

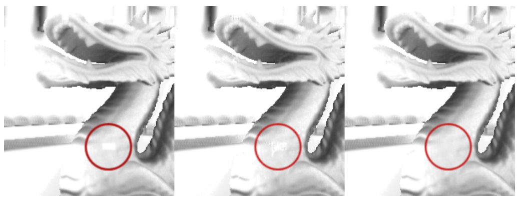

**Figure 7:** *(left) VAO with 'greedy' culling pre-pass. Notice white square artifact. (center) 'Careful' pre-pass significantly improves result. (right) Reference with no culling pre-pass. Images have been gamma corrected to exaggerate the artifact.*

## 4.3. Luminance Sensitivity

One of the common problems with SSAO methods is that they only consider geometrical scene information. As such they usually draw occlusion over surfaces where no ambient shadows should appear – such as light sources and emmisive or highly specular materials. To counteract these defects we propose a *luminance sensitivity* extension. The luminance sensitivity suppresses ambient occlusion based on the brightness (luminance) of the evaluated surface (Figure 8).

The method uses a user-specified brightness threshold above which the AO component is going to be reduced. For given point $x$ the luminance corrected ambient occlusion $AO_L(x)$ is expressed as:

$$AO_L(x) = AO(x)\left(1 - \frac{min(1, max(0, lum(x) - threshold + width))}{2width}\right)^{\sigma}, \tag{3}$$

where $AO(x)$ is the input ambient occlusion, $lum(x)$ expresses the luminance computed from RGB color of $x$, *threshold* is a user specified threshold, and *width* and *sigma* are constants defining the profile of the correction function. Alternativelly we use the $V$ component of the $HSV$ color model instead of luminance, which provides results less dependent on the hue color component. Examples of luminance snsitivity curves are shown in Figure 9.

## 4.4. Depth Range Check

One of the biggest disadvantages of using the depth buffer as an approximation of the scene geometry lies in loss of information about the occluded parts of the scene. Without any additional source of information the VAO algorithm considers all occluders infinitely thick (see the Case C of the VAO method). Note that this causes undesired shadow outline in areas where no occlusion should occur (Figure 10). There had been many proposals on how to counteract this issue to produce realistic results in the past such as depth peeling [Eve01] or getting the information from other camera views [VPG13]. These solutions however introduce heavy performance overhead. We propose a fast method that produces visually satisfactory results and requires no additional settings from the user.

Instead of considering occluders infinitely thick, we assume they are flat and their thickness gradually decreases. When given sample is fully inside geometry (VAO Case C), we decrease occlusion depending on distance from occluder which is expressed by the *falloff function* (Equation 4). From numerous tested functions we have selected one that yielded aesthetically best results. This function essentially describes the thickness of geometry behind what is seen by the camera.

$$s(d) = \frac{u}{max(u, \frac{d}{radius})} \tag{4}$$

$s(d)$ expresses occlusion contribution of the sample that is in distance $d$ behind occluder in view-space units taking into account the kernel *radius*. The weight of the sample (length of the pipe intersecting the tagent sphere) is reduced by multiplying it by $s(d)$ to

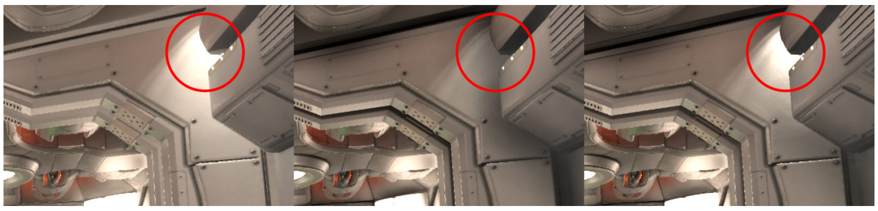

**Figure 8:** *Scene without AO, AO without luminance sensitivity and AO with the luminance sensitivity enabled.*

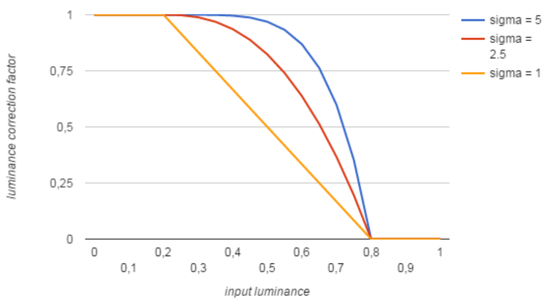

**Figure 9:** *Luminance sensitivity curves for various sigma (linearity) settings. The threshold was set to 0.5 and width to 0.3.*

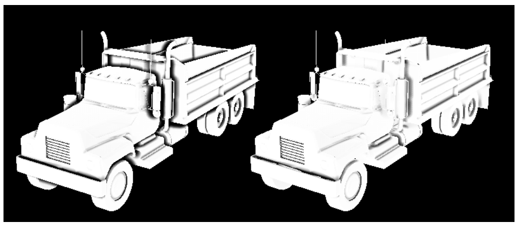

**Figure 10:** *(left) Visible shadow outline in original algorithm. (right) Corrected result with depth range check.*

gradually decrease occlusion with increasing distance. The $u$ parameter offsets thickness by certain amount to suppress artifacts (self-occlusions) caused by numerical imprecision of depth buffer (Figure 11). We use empirically selected value of 0.03 for $u$ in our implementation which creates smooth transition in areas with depth discontinuity and suppresses the halo artifact significantly.

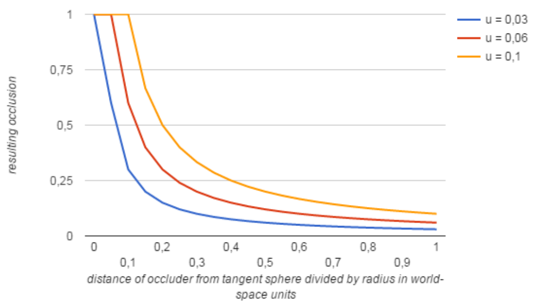

**Figure 11:** *Falloff function describing the thickness of geometry seen by the camera.*

## 4.5. Samples Generation

One of the most significant factors on final visual result is the quality of sample sets used to estimate the occlusion volumes. The VAO algorithm uses samples distributed within the disk of tangent sphere great circle as opposed to more conventional SSAO solutions which use three-dimensional distributions of samples in the hemisphere.

We employ *interleaved sampling* by rotating the sample set in each pixel by uniformly distributed angle to get a different (yet symmetrical) sample set. We use repeated pattern of rotation angles in $3 \times 3$ pixel tiles. Experimentally we have established that the $3 \times 3$ tile size is sufficient to produce results of high visual quality (the number of samples is virtually increased by factor of 9) and small enough to be completely removed by fast low-pass filter of matching size. Therefore we require to generate a sample set that yields a uniform sample distribution after rotating it nine times.

We use a sample generator based on Poisson disk distribution that generates a compact sample set that considers the rotations during generation. The main goal of this method is the reduction of the banding artifacts caused by uneven distribution of samples (see Figure 12).

Our Poisson Disk sample generator takes the future rotations of samples into account (Figure 13). We use nine uniform-step angles for rotations. This creates a minor advantage in that rotations do not have to be fetched from a custom texture but rather can be calculated from UV coordinates (this resulted in about 1% performance increase). Our generator stores not only newly added points, but also all of their rotations. When another point is being considered as a sample candidate, this approach ensures that it will be far enough from all the other points including their rotations.

After generating the Poisson disk sample distribution, we select the best permutation of rotated samples so that sum of their distances from each other is maximal. This further lowers the noise introduced by rotations. For sample-sets of size 8 and less it is feasible to test all permutations. For larger sets, we incrementally add groups of 8 samples and look for best permutation of last group with fixed previous groups (Figure 14).

Finally, samples are sorted according to the *z-order*, so that samples spatially close in the scene are also close to each other in memory. This increases cache coherence while fetching the data from depth and normal textures.

The sample set generated with the above described algorithm also improves the visual quality of the transition between the sets of different sizes in the adaptive sampling step of the method.

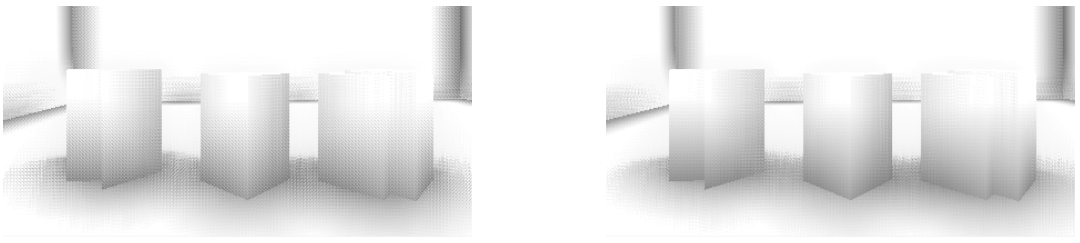

**Figure 12:** *Reduction of banding artifacts by taking the sample rotations into account during their generation. (left) Rotation-unaware Poisson disk distribution. (right) Using the sample set generated by the proposed method.*

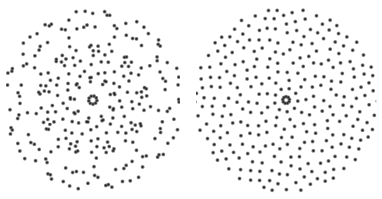

**Figure 13:** *Comparison between the uniform distribution that is rotated afterwards and our modified distribution that considers rotations during generation (16 samples multiplied by 9 rotations).*

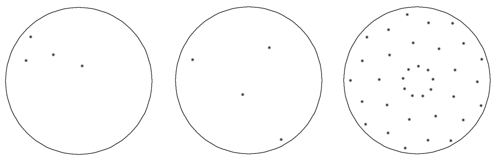

**Figure 14:** *Sampling kernel of size 4 generated by Poisson Disk generator (left). Best permutation of rotations (middle). All rotations (right).*

## 4.6. Low Pass Filter

When using lower number of samples to calculate AO, more artifacts caused by undersampling will occur. This happens for 2 and 4 samples (and depending on the radius sometimes even for 8 samples). These artifacts display as occlusion with jagged edges as low number of samples are not enough to sample scene geometry properly. To remove these artifacts we provide optional gaussian blur of variable size. The user can set size of convolution kernel and deviation of gaussian bell curve to fine-tune the balance between amount of artifacts and 'blurriness'.

We use a non-linear cross-bilateral filter: we do not take neighbouring pixels with depth variation over a certain pre-defined threshold into account. As the threshold we take AO radius in world/view space units and calculate linear view space depth for each pixel. Omitting depth-awareness part causes disturbing halo artifacts on areas with depth discontinuities even for convolution of this small size.

Because the convolution filters of larger kernel sizes are performance-intensive, we implement the filter in two passes as if the filter was separable (though technically it is not). It produces very good results, even in the worst-case scenario when a depth discontinuity splits filter on the kernel diagonal.

## 4.7. Color Bleeding

As with some other SSAO methods we also use a simple simulation of color bleeding using an approximation of one indirect light bounce. To add this effect, the algorithm samples not only the geometry (represented by depth and normals), but also the color of the surface. We calculate the contribution of each sample based on the configuration of the sample position relative to shaded surface and it's normal. In particular we compute the amount of indirect contribution of sample $x$ to shaded point $p$ as:

$$c(x) = max(0, \frac{(x-p)\cdot n}{|x-p|})(1 - \frac{|x-p|}{radius})^2 \qquad (5)$$

where $n$ is the surface normal and $radius$ is the radius of the AO kernel. We use $c(x)$ to linearly interpolate between the white color and the sampled color.

The problem with reusing the same samples for the secondary bounce arises from the fact that the sample can spatially lie deep inside the geometry – but for correct results we would like to sample the geometry surface. This would require ray marching along the ray from shaded surface to sample the position until we hit the surface. Such an operation is performance intensive and impractical. Another solution is to discard samples that do not lie close to the surface. However, this discards too many samples to provide usable results.

In our method, we relax this requirement - we use all samples regardless of whether they lie on the surface or not. We rely on the fact that problematic samples will occur when there is some surface close to ray's origin – this means that their contribution will be hidden by a strong ambient occlusion in that area. This yields good results and performance. To improve the performance further, we can use either half or quarter of the samples to calculate color bleeding simply by taking every second or fourth sample. Because samples are z-ordered, we will still sample the whole circle area evenly.

## 5. Unity Implementation

VAO++ can be implemented into rendering solutions that generate per-pixel normals and provide access to depth buffer (any modern deferred renderer). Effect is applied to resulting image simply by multiplication by occlusion values. This is commonly achieved as an additional rendering pass taking renderer result as input and producing new result to screen. We have chosen Unity for its popularity and ease of use.

Unity was designed to be extendable with custom post-processing effects. The effects can be implemented either as an *image effect* that is usually attached to the camera node in the scene graph, or as a *command buffer* that is a list of graphics commands that extend existing rendering pipeline. Our implementation provides both options for solving compatibility issues on certain deployment platforms.

### 5.1. Image Effect Implementation

The implementation of VAO++ as an image effect in Unity is very intuitive as the effect has similar behavior to other Unity components. The user has a full control over the order in which the image effects are applied. The implementation consists of a pair of shader and script files. The script loads and executes the shader when *OnRenderImage* event is raised. It also contains GUI controls and sends the selected settings onto the GPU via shader uniform variables. The shader file contains the implementation of the image effect as vertex and fragment shaders organized in *passes*. One of the problems arising from the image effect approach is that it is applied as a post processs after all rendering passes are finished. This can place ambient occlusion on transparent primitives, over previous effects, and if HDR rendering is enabled, also on brightly lit surfaces.

### 5.2. Command Buffer Implementation

The implementation of VAO++ using a command buffer is more complicated, but the effect can be inserted into the rendering pipeline where it is supposed to be, i.e. after all opaque primitives are rendered and before lighting is calculated. This will however make harder to control the order in which the image effects are applied to the given scene. The implementation shares a shader file with the image effect and when enabled, it bypasses the *OnRenderImage* event. Instead, it fills the command buffer in the *OnPreRender* event and sends it for processing within the rendering pipeline. The command buffer itself contains commands to create temporary textures and perform certain shader passes as needed by the algorithm.

### 5.3. VAO++ Shader

We started with an implementation of prototype in OpenGL/GLSL to eliminate any platform-specific quirks and fine-tuned the algorithm there. Then we rewrote the shader into Cg/HLSL language used by Unity.

Unity provides access to G-Buffer via special texture *_CameraDepthNormalsTexture* that contains octa-encoded view-space normals in the first 16 bits and the linear depth packed into the remaining 16 bits. Decoding compressed and packed depth and normals is done via built-in macro *DecodeDepthNormal*. Because the depth is already linearized and in <0,1> range, there is no need to recalculate it from reciprocal value as in most other implementations. Because depth and normals are compressed into 16 bits each, their precision is lower than precision available on modern GPUs. Thanks to it's design, the algorithm is quite resistant to these imprecisions and it doesn't require a magic bias variable commonly found in similar effects.

The view space position of shaded pixel is calculated by reprojection. In the vertex shader, the position of each screen corner is reprojected from screen space into view space using the inverse of projection matrix (pre-calculated on the CPU). The resulting values from the vertex shader are interpolated automatically and in the fragment shader, they represent a directional vector from the camera onto the far plane in given pixel. This vector is normalized and multiplied by depth value to get the view-space position of given pixel.

## 6. Results and Discussion

In this section we present the impact of different VAO++ extensions on the algorithm performance (Table 1). All measurements were run on the NVIDIA GTX 970 graphics card and measured with Unity Profiler. Both of our major optimizations are measured separately to better show the extent of their effect on different image resolutions and sample counts. We also compare our results with the Unity SSAO contained within the Unity Post-processing Stack plugin. Finally, we include sample images of different scenes to show the behavior of the method for different distances and geometry (Figure 17).

The culling pre-pass gains performance for sample sets of size 8 and higher. This is caused by the fact that occlusion for culling candidates is always calculated with sample count of 8. The down-sampling nature of this algorithm also scales the speedup for higher resolutions – the culling pre-pass achieved 30% speedup in 1080p and almost 40% in 4k resolution.

The performance boost caused by the adaptive sampling progressively increases with sample set size. Higher numbers of samples give greater potential for sample count reduction at farther distances.

Table 2 and Figure 15 contain the comparison between VAO++ and Unity SSAO. Both methods were set in such a way as to make the AO shadows of approximately the same size. For 16 samples VAO++ is about 20% faster than Unity SSAO in 1080p and over 30% faster in 4k. It is also worth noting that further doubling the size of the VAO++ sample set (to 32 samples) is only 15% slower than Unity SSAO with 16 samples in 1080p and just 3% slower in 4k.

Visual results of VAO++ are close to raytraced AO (Figure 16) and it preserves shapes of generated shadows also when moving camera and for dynamic objects. VAO++ does not require large low pass filter like Unity's SSAO which results in sharper images and shadows that match the geometry of the scene better and look more realistic. We also employ more aggressive range check function which we believe yields better results and does not over-darken the scene unnecessarily.

Our extensions aimed at improving performance have very low

| | 1920×1080 | | | | 3840×2160 | | | |
|---|---|---|---|---|---|---|---|---|
| Sample Count [-] | VAO [ms] | Adaptive Sampling [ms] | Culling Pre-pass [ms] | VAO++ [ms] | VAO [ms] | Adaptive Sampling [ms] | Culling Pre-pass [ms] | VAO++ [ms] |
| 2 | **0.65** | 0.75 (115%) | 0.88 (135%) | 0.93 (143%) | **2.75** | 3.14 (114%) | 3.65 (133%) | 3.88 (141%) |
| 4 | 0.93 | **0.92 (99%)** | 1.04 (111%) | 1.02 (110%) | 4.08 | **3.91 (96%)** | 4.31 (106%) | 4.26 (104%) |
| 8 | 1.53 | 1.27 (83%) | 1.37 (89%) | **1.2 (78%)** | 6.87 | 5.53 (80%) | 5.77 (84%) | **4.97 (72%)** |
| 16 | 2.67 | 1.88 (70%) | 2.04 (76%) | **1.59 (59%)** | 12.74 | 8.24 (65%) | 8.67 (68%) | **6.59 (52%)** |
| 32 | 5.04 | 3.09 (61%) | 3.38 (67%) | **2.33 (46%)** | 23.71 | 13.76 (58%) | 14.47 (61%) | **9.77 (41%)** |

**Table 1:** *Time spent computing the effect with different optimizations. Percentages are the running time ratios with respect to the unaccelerated VAO. The fastest setting is shown in bold.*

| Sample Count | | 1920×1080 | | | 3840×2160 | | |
|---|---|---|---|---|---|---|---|
| VAO++ | SSAO | VAO++ [ms] | SSAO [ms] | Ratio [%] | VAO++ [ms] | SSAO [ms] | Ratio [%] |
| 2 | 3 | 0.65 | 0.73 | 89 | 2.75 | 3.62 | 76 |
| 4 | 6 | 0.92 | 1.05 | 88 | 3.91 | 5.07 | 77 |
| 8 | 10 | 1.2 | 1.43 | 84 | 4.97 | 6.82 | 73 |
| 16 | 16 | 1.59 | 2.01 | 79 | 6.59 | 9.54 | 69 |

**Table 2:** *The best VAO++ result compared to Unity SSAO implementation for the scene on Figure 15. Note that both algorithms use different sample counts for each quality level.*

impact on visual quality – culling pre-pass causes artifacts of negligible size and adaptive sampling only affects areas further from camera. Samples generation is run offline and results are reused. Therefore our extensions do not impose any additional limitations compared to original VAO algorithm.

## 7. Conclusion

We described the VAO++ method that extends the previously proposed VAO algorithm [SKUT*10] with several optimizations improving its performance and visual quality. These optimizations include adaptive sampling, culling pre-pass, luminance sensitivity, depth range check, and efficient interleaved sampling with a compact sample set. We described the implementation of the method within the Unity game engine including two different ways of integrating the method into the rendering engine.

We evaluated the method in terms of speed and visual quality. The results show that the method achieves up to 30% faster rendering while keeping the same quality as the standard Unity implementation of SSAO. In the future we plan to extend the method by further optimizing the sample set using a kernel with variable sampling density together with multi-scale depth buffer representation.

## Acknowledgements

We thank Marko Dabrovic for the Sibenik scene, Anat Grynberg and Greg Ward for the Conference scene, Stanford University for the Dragon model, and Unity Technologies for Corridor and Viking Village scenes. This work was partially supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS16/237/OHK3/3T/13.

## References

[BE05] Bunnell M., Elements S.: Dynamic ambient occlusion and indirect lighting. *GPU Gems* (2005), 223–233. 2

[CAM08] Clarberg P., Akenine-Möller T.: Exploiting Visibility Correlation in Direct Illumination. *Computer Graphics Forum (Proceedings of EGSR 2008) 27*, 4 (2008), 1125–1136. 2

[Chr08] Christensen P. H.: *Point-Based Approximate Color Bleeding*. Tech. Rep. 08-01, Pixar, Emeryville, CA, 2008. 2

[Eve01] Everitt C.: Interactive order-independent transparency. White paper, nVIDIA, 2(6), 7, 2001. 4

[HJ08] Hoberock J., Jia Y.: High-quality ambient occlusion. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley, 2008, pp. 257–274. 2

[IKSZ03] Iones A., Krupkin A., Sbert M., Zhukov S.: Fast, realistic lighting for video games. *IEEE Computer Graphics and Applications 23*, 3 (2003), 54–64. 1, 2

[KA06] Kontkanen J., Aila T.: Ambient occlusion for animated characters. In *proceedings of Eurographics Symposium on Rendering* (2006), pp. 343–348. 2

[KL05] Kontkanen J., Laine S.: Ambient occlusion fields. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics* (2005), pp. 41–48. 2

[Lan02] Landis H.: Production-ready global illumination. acm siggraph course, 2002. 1, 2

[Mit07] Mittring M.: Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 Courses* (New York, NY, USA, 2007), SIGGRAPH '07, ACM, pp. 97–121. 1, 2

[MSC03] Mendez A., Sbert M., Cata J.: Real-time obscurances with color bleeding. In *Proceedings of Spring Conference on Computer Graphics* (April 2003). 2

[Ngu07] Nguyen H.: *Gpu Gems 3*, first ed. Addison-Wesley Professional, 2007. 2

[Pho75] Phong B.-T.: Illumination for Computer Generated Pictures. *Communications of ACM 18*, 6 (1975), 311–317. 2

[RGS09] Ritschel T., Grosch T., Seidel H.-P.: Approximating Dynamic Global Illumination in Screen Space. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2009). 2

[SA07] Shanmugam P., Arikan O.: Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 73–80. 1, 2

[Sai08] Sainz M.: Real-time depth buffer based ambient occlusion. gdc presentation., 2008. 2

[SKUT*10] Szirmay-Kalos L., Umenhoffer T., Toth B., Szeci L., Sbert M.: Volumetric ambient occlusion for real-time rendering and games. *IEEE Computer Graphics and Applications 30*, 1 (Sept. 2010), 70–79. 2, 3, 8

[VPG13] Vardis K., Papaioannou G., Gaitatzes A.: Multi-view ambient occlusion with importance sampling. In *Proceedings of the Symposium on Interactive 3D Graphics and Games* (2013), pp. 111–118. 4

[ZIK98] Zhukov S., Iones A., Kronin G.: An ambient light illumination model. In *Rendering Techniques '98: Proceedings of the Eurographics Workshop* (1998), pp. 45–55. 1, 2

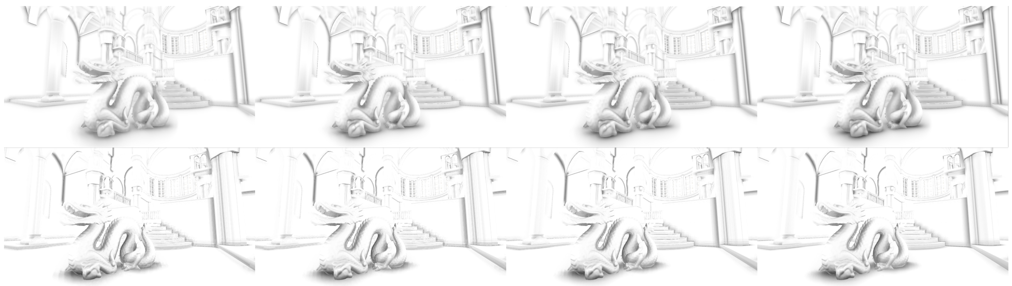

**Figure 15:** *The comparison of Unity SSAO (top) and VAO++ (bottom) for increasing sample counts. For Unity SSAO we used 3, 6, 10, and 16 samples, for VAO++ 2 4, 8 and 16 samples.*

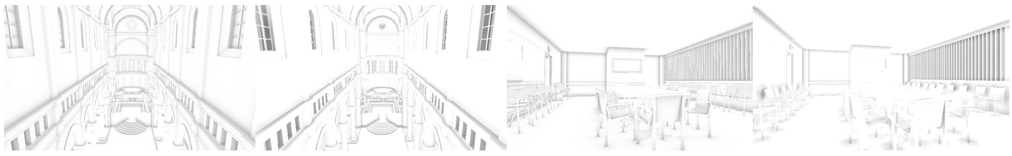

**Figure 16:** *Comparison of VAO++ to ray-traced reference. Left to right: Ray-traced AO Sibenik, VAO++ Sibenik, Ray-traced AO Conference, VAO++ Conference.*

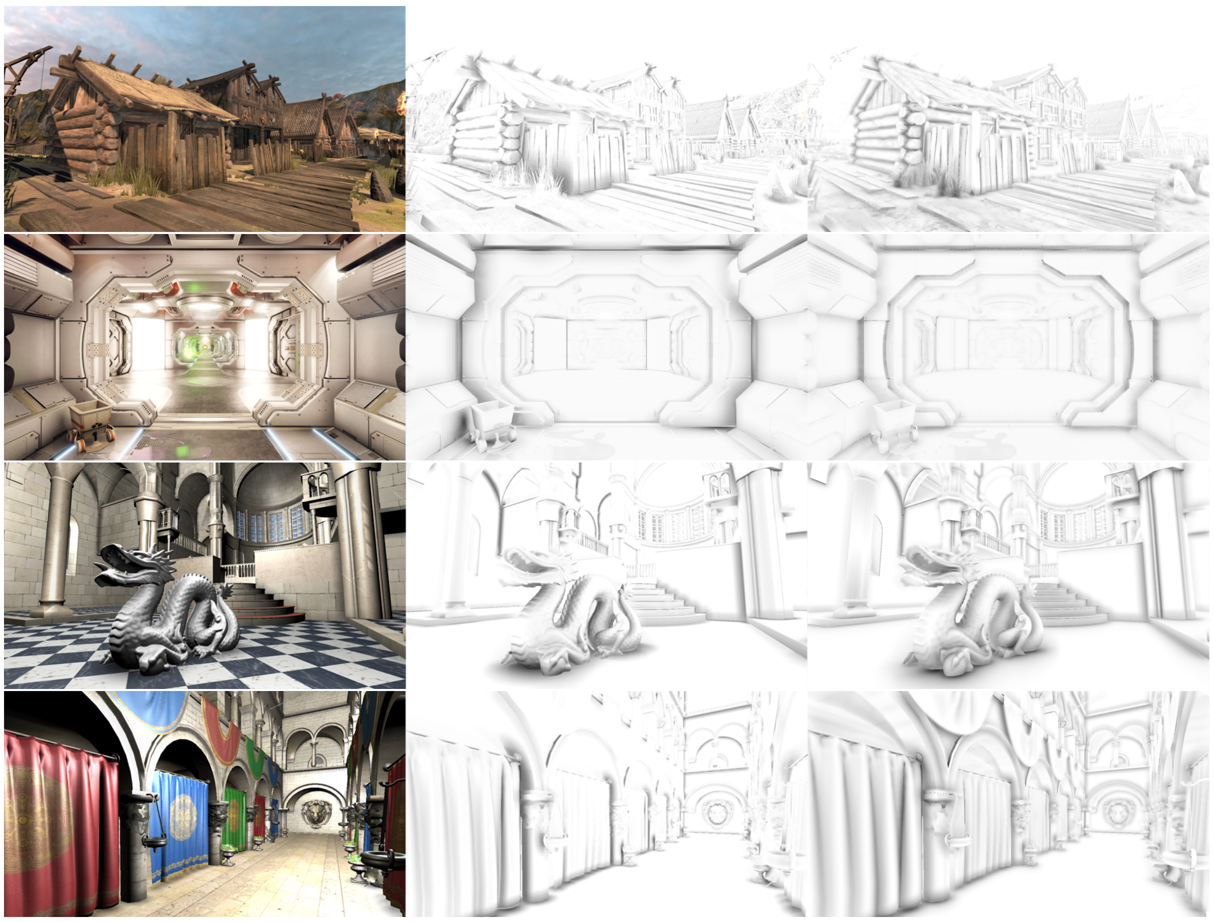

**Figure 17:** *(left) Snapshots of the test scenes (Viking Village, Corridor, Sibenik and Sponza) rendered with VAO++, (center) VAO++ visualization, (right) Unity's built-in SSAO visualization.*