Line-Sweep Ambient Obscurance

Ville Timonen[†]

Turku Centre for Computer Science
Åbo Akademi University

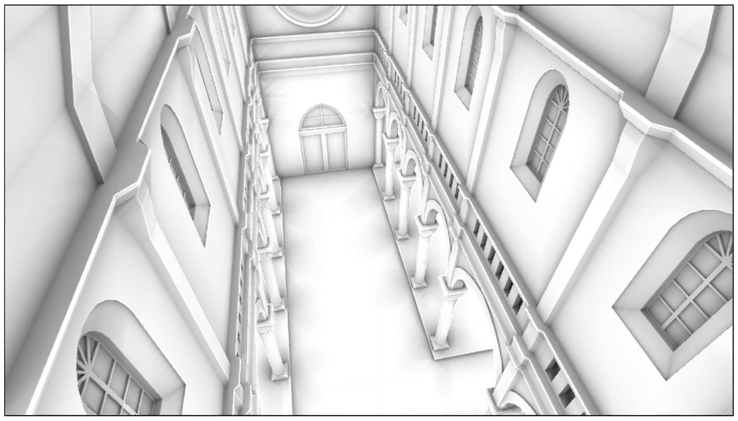


Figure 1: *SSAO rendered by our method at 1920×1080 (+10% guard band) in 1.7 ms on a GeForce GTX 480.*

Abstract
Screen-space ambient occlusion and obscurance have become established methods for rendering global illumination effects in real-time applications. While they have seen a steady line of refinements, their computational complexity has remained largely unchanged and either undersampling artefacts or too high render times limit their scalability. In this paper we show how the fundamentally quadratic per-pixel complexity of previous work can be reduced to a linear complexity. We solve obscurance in discrete azimuthal directions by performing line sweeps across the depth buffer in each direction. Our method builds upon the insight that scene points along each line can be incrementally inserted into a data structure such that querying for the largest occluder among the visited samples along the line can be achieved at an amortized constant cost. The obscurance radius therefore has no impact on the execution time and our method produces accurate results with smooth occlusion gradients in a few milliseconds per frame on commodity hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Computer Graphics—Color, shading, shadowing, and texture

1. Introduction

Ambient occlusion and obscurance (AO) have become de-facto parts of global illumination implementations, and their screen-space evaluation (SSAO) has been widely adopted in real-time applications. Since its inception in 2007, SSAO has seen a steady line of improvements both in render quality and render time. However, its computational complexity has remained fundamentally the same. The distance of the AO effect is defined in eye-space and may thus cover a large range in screen-space, and high quality results still require

[†] e-mail: vtimonen@abo.fi

more samples per pixel than is practically affordable in real-time applications.

One established SSAO method is Horizon-Based Ambient Occlusion (HBAO) [BSD08] [Bav11] which accumulates obscurance from a set of azimuthal directions, finding the largest occluder in each direction by ray marching. While HBAO's physically based treatment of geometry scales well quality-wise, the number of samples that is necessary in each azimuthal direction for high quality results is prohibitive performance-wise. Given K azimuthal directions and N steps along each direction, HBAO's time complexity per pixel is $O(KN)$.

We propose a method to calculate the same results in $O(K)$ time with unlimited range. Instead of calculating the obscurance independently for each receiver pixel, we perform line sweeps over the screen. Each line is traversed incrementally, and the visited geometry along the line is stored in an internal data structure which can be queried in amortized constant time for the largest falloff attenuated occluder at the new pixel. This significant reduction in the time complexity of SSAO allows the fast production of high quality renderings which is not impacted by the range of the effect. Since we are not allowed to choose the azimuthal directions for each pixel freely, but rather use a set of directions shared by multiple screen pixels, our main visual artefact is banding which can be alleviated by increasing K.

2. Previous Work

Evaluating AO from the geometry in the depth buffer was first proposed by [Mit07] and [SA07] who sample a volume around the receiver pixel and determine the occlusion from the number of points that fall below the depth field. As scene geometry not merely blocks incoming light but also reflects it, an empirically selected falloff function [ZIK98] is usually introduced that weighs the sampled scene points according to their distance from the receiver by putting more weight to nearby occluders. Ambient occlusion extended by a falloff function has been termed ambient *obscurance*. Since [Mit07] and [SA07], several works such as [BS09] [LS10] [MOBH11] [MML12] have refined the quality and rendering speed of SSAO methods.

Evaluating each sampled scene point independently from each other ignores the fact that the evaluation of occluders in roughly the same azimuthal direction is not separable: A tall nearby occluder might make occluders behind it invisible such that these do not contribute to occlusion regardless of their elevation. To address this issue [BSD08] takes a more physically based approach along the lines of Horizon Mapping [Max88], whereby the highest horizon within a certain range is searched for a set of azimuthal directions. While this approach scales well with respect to image quality and obscurance radius, it is expensive because many height field samples have to be taken in each azimuthal direction. Our method produces results essentially identical to [BSD08] but we find the largest occluder in $O(1)$ time for one azimuthal direction within an unbounded radius.

We build upon the observation by [TW10] that sweeping through a height field in lines and incrementally building the convex hull of the visited geometry allows the extraction of global horizons in constant time for the purpose of horizon mapping. In ambient obscurance, where the falloff function attenuates occlusion with distance, the global horizon is often very far away and contributes little to occlusion, making the convex hull of little use in determining AO. Also, [TW10] scans the height field densely and accumulates rotated versions of the sweeps. This results in banding unless many azimuthal directions are scanned, which in turn becomes computationally expensive.

Our contribution over [TW10] is three-fold:

- Instead of using a geometrical convex hull, we form a hull based on the falloff weighted obscurance.
- We generalize the scans to arbitrary sampling densities and propose a way to gather results sparsely per-pixel, which allows trading high render times for edge-respecting blur when the number of azimuthal scanning directions is increased.
- We also suggest special line sampling patterns for cases where depth buffer values cannot be interpolated (often the case in SSAO).

3. Overview

The observation behind HBAO is that SSAO is physically separable per pixel in azimuthal directions. We also exploit this observation and calculate obscurance in K discrete azimuthal directions. For physically correct ambient obscurance, in each azimuthal direction the falloff weighted occlusion should be integrated from the tangent plane upwards until the global horizon as shown to the left in Figure 2. This results in unavoidably high complexity as obscurance has to be gathered in many segments.

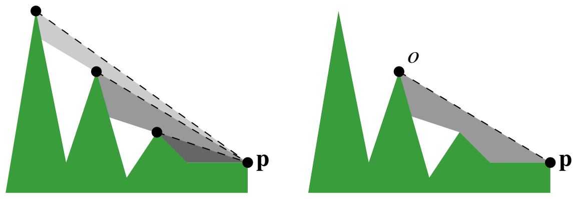


Figure 2: *Left: Obscurance gathered in segments of the elevation angle between the tangent plane and the global horizon onto receiver* **p**. *Each segment is shaded according to the strength of the falloff term. Right: Obscurance from the largest falloff weighted occluder, o, only.*

However, if we are willing to make the sacrifice that we calculate obscurance in each azimuthal direction from only one occluder along that direction—the one that casts the largest obscurance, as shown to the right in Figure 2—an order of magnitude more headroom is made available complexity-wise, which we in this paper show how to exploit. We define *largest occluder* as the occluder that would cast the largest amount of obscurance on the receiver were it the only occluder along the azimuthal direction. Fortunately, it turns out that the visual impact of only considering the largest occluder per direction is modest, as illustrated in Figure 3.

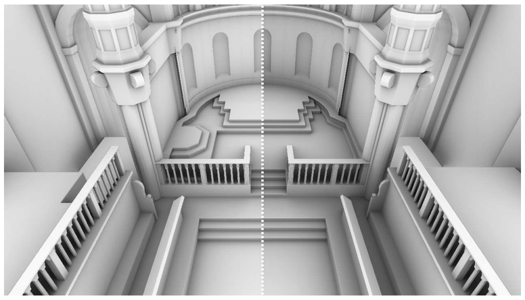


Figure 3: *Right: Full obscurance. Left: Obscurance from the largest falloff weighted occluder only.*

Approximating obscurance from a single occluder typically yields underestimated obscurance because geometry below the largest occluder tends to be closer than the largest occluder (higher falloff term) and also because obscurance coming above the largest occluder is simply ignored. However, given the approximated nature of SSAO the results are entirely plausible and can be further compensated by lifting the falloff function or by adjusting brightness/contrast in post-process.

3.1. Our Method

Instead of calculating AO for each receiver pixel independently (done predominantly in prior work) we traverse through the depth field along lines that cover the framebuffer evenly as shown in Figure 4. In a threaded implementation

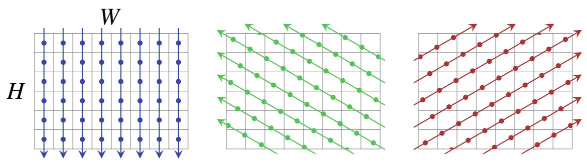


Figure 4: *Sweeps in $K = 3$ directions over a 8×6 framebuffer. A total of $K \cdot W \cdot H$ samples/receivers are considered.*

each thread processes one line, and the line is traversed one constant length step at a time. At each step the depth field is sampled and the sampled point is deprojected into eye-space. This point is then stored onto a stack using an algorithm (described in detail in Section 4) that has an amortized constant cost per step. Processing a line of N steps therefore has the complexity of $O(N)$. At any given point the largest occluder is always found at the top of the stack.

Our method is compatible with obscurance estimators that evaluate obscurance from a set of azimuthal directions and as input take the horizon angle and the distance to the occluder. In this paper we have chosen to use HBAO's obscurance estimator:

$$AO(\mathbf{p},\vec{n}) \approx \frac{1}{K}\sum_{k=0}^{K-1}\left(sin(t) + (sin(h) - sin(t))\,\rho(||\vec{h}_k||)\right), \quad (1)$$

$$\vec{D}_k = (sin(\alpha), cos(\alpha)), \alpha = k \cdot 2\pi/K,$$
$$\vec{u}_k = -(p_{xy} \cdot \vec{D}_k, p_z), \vec{t}_k = (-\vec{n}_z, \vec{n}_{xy} \cdot \vec{D}_k), \vec{h}_k = (\vec{o}_{xy} \cdot \vec{D}_k, \vec{o}_z),$$
$$sin(t) = \hat{t}_k \cdot \hat{u}_k, sin(h) = \hat{h}_k \cdot \hat{u}_k$$

for receiver point $\mathbf{p}$ with normal $\vec{n}$. The eye is located at the origin and $\vec{u}_k$ is the zenith vector towards the eye. All vectors denoted by subscript k are projected onto the 2D plane along the azimuthal direction defined by the vector $\vec{D}_k$. Vector $\vec{o}$ points from $\mathbf{p}$ towards the largest occluder along the azimuthal direction. The terms are illustrated in Figure 5. It should be noted that the first $sin(t)$ in the sum in Eqn. 1 gets canceled by opposing directions when both directions use the same tangent plane. As we discuss strategies where this is not the case, in Section 4.1, we include the term in our obscurance estimator to ensure that AO does not evaluate to a value larger than 1.

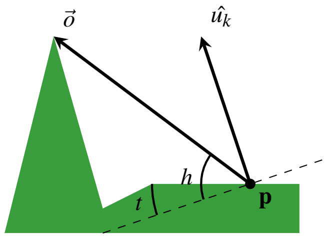


Figure 5: *Illustration of the terms used by our obscurance estimator in Eqn. 1. Vector $\vec{o}$ is the vector from the receiver* $\mathbf{p}$ *to the largest occluder, $\hat{u}_k$ is the unit zenith vector towards the eye, and t and h are the tangent plane and horizon angles, respectively, from the zenith normal.*

Our method can use any monotonically decreasing falloff function ρ and for this paper we have selected an inverse quadratic function similar to the function reported aesthetically agreeable in [FM08]:

$$\rho(d) = \frac{r}{r + d^2} \quad (2)$$

where r is used to control the decay rate.

The obscurance results written along the processed lines are gathered per pixel in a separate phase, described in Section 5. In Section 5 we also cover how lines are positioned in the framebuffer and how sampling coordinates should be chosen. Rendered images and execution times are then presented in Section 6 and compared against the most relevant previous work.

4. Line Sweeping

In this section we describe the process of sweeping through one line in the framebuffer. The output of this process are obscurance values for the points along the line written to an intermediate buffer.

4.1. Obscurance Hull

We first recapitulate the main idea behind *incremental convex hulls* as introduced in [TW10] as our data structure is motivated by it. In [TW10] height field points are iterated along a line and incrementally inserted onto a stack that holds the convex subset of the visited points. In order to keep the stack convex before pushing a new point in, elements are popped from the stack (shown in red in Figure 6) until the last 2 points on the stack and the point to be added form a convex set. The three points form a convex set when the vector from the new point to the last point on the stack has a lower slope than the vector from the new point to the point second to last in the stack. Because the stack can be assumed to have been convex in the previous iteration, due to induction it will remain convex after pushing the new point in. In the convex hull the global horizon for the new point is cast by the point next to it, which is now second to last in the hull (shown in blue in Figure 6).

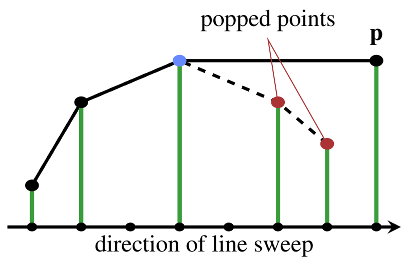


Figure 6: *A convex hull before (dashed line) and after (solid line) adding the new point* **p**. *The global horizon for* **p** *is cast by the point next to it in the hull, marked in blue.*

For the purpose of determining SSAO however, the global horizon might be far away from the receiver, and, according to the falloff function, might cast an insignificant obscurance on the receiver. Since we are trying to find the point between the global horizon and the receiver that casts the largest *falloff weighted* occlusion we have to use an additional criterion to geometrical convexity for the hull. Instead of popping the stack until the point to which the slope from the receiver is the lowest is at the top, we pop the stack until the point which casts the largest obscurance onto the receiver is at the top. The main difference to [TW10] is the boolean function (with the three points as parameters) which determines whether to pop elements from the stack: Instead of testing for convexity, we compare the obscurance (Eqn. 1) from the last 2 points in the stack onto the new point. Points are popped from the stack until the last points casts the largest obscurance, or until the global horizon is reached. We refer to a hull formed according to these criteria as an *obscurance hull*. The obscurance hull will not necessarily be convex and points in the beginning of the stack are progressively losing weight due to the falloff function.

However, it is possible that in some extreme cases the obscurance hull does not return the largest occluder for a receiver. In order to provide some intuition on when this can happen, consider ρ to be a step function that puts full weight to occluders within distance r and zero to others. Next, consider that the colored points in Figure 6 are within r from **p** and a new point, **q**, is encountered after **p** along the line. Now, let **q** be at a height where the popped (red) points are visible to **q** and within r, whereas the blue point falls outside r. In this case one of the popped points casts the largest obscurance on **q** but instead **p** is returned by the obscurance hull. Cases like this are rare but can occur in areas of rapid depth changes. We measured the average error in the final AO value caused by these degenerate cases to be small—between 0.02% and 0.3% in scenes presented in this paper.

Some attention must be paid to the fact that the tangent plane of the receiver appears in Eqn. 1. Obscurance is cut by $sin(t)$ which is not globally fixed. Therefore forming an obscurance hull using one tangent plane might not give the correct obscurance onto a receiver that has a different tangent plane. Eqn. 1 will be used to determine which points are to be culled atop the stack, and also for calculating the obscurance on the receiver once the largest occluder is found. There are three main strategies for choosing the tangent plane for these two operations:

1. Points are culled and obscurance is calculated using the receiver's real tangent. This however will cause the obscurance hull to rapidly unfold when the tangent becomes steep (high $sin(t)$), which may result in future largest occluders being culled and therefore in missing occlusion.
2. Points are culled assuming a globally fixed tangent while the obscurance is calculated using the receiver's real tangent. This makes occlusion discontinued because the obscurance hull gives the largest occluder based on different criteria than the actual per-receiver obscurance is calculated with.
3. The tangent plane is fixed for all calculations ignoring the real tangents of the points.

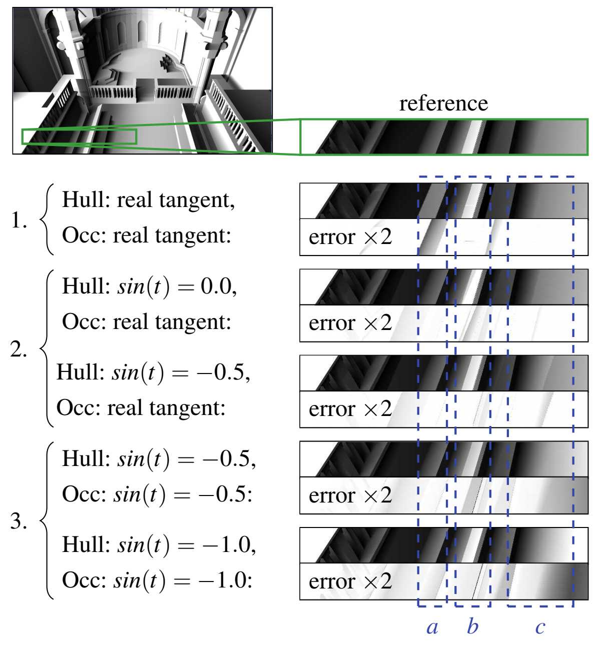


Figure 7: *Various strategies for treating the tangent plane $sin(t)$ when constructing the obscurance hull ("Hull:") and when calculating the obscurance ("Occ:"). The images show the contribution of a single AO sweep directed to the right. A cut from the Sibenik scene is shown for each strategy above the respective error image. Areas of interest a, b, and c are highlighted in blue.*

Figure 7 visualizes ambient obscurance contribution from a single sweep in the Sibenik scene using the three strategies. A cut from the scene is shown from a part where the tangent varies with respect to the sweep direction. The reference image is created by going through all steps along the sweep line for each receiver individually in brute-force and picking the largest occluder.

Strategy 1 may result in occlusion that is flat in regions where, instead, a gradient should appear (area *a* in Figure 7). Along a sweep, strategy 2 may switch from an occluder to the next at a point where occlusions cast by the consecutive occluders do not match. This causes a jump in the occlusion value which is perceptually prominent (area *c* in Figure 7). Strategy 3 both constructs the hull and evaluates occlusion using the same tangent value and therefore produces consistent, however biased, obscurance. The bias comes from the fact that the tangent plane splits the occlusion integral in two parts whose falloff terms differ: The first part is below the tangent plane and has $\rho = 1$ and the second is above the tangent plane and has $\rho \leq 1$ that is dependent on the distance to the occluder. If $sin(t)$ is chosen large, part of the integral that should be evaluated using the occluder's distance (i.e. is above the real tangent) may instead be considered to be below the fixed tangent plane and evaluated using $\rho = 1$ causing overestimated occlusion. This is visible in area *b* when $sin(t) = -0.5$. If $sin(t)$ is chosen small, the opposite may happen: Integral below the real tangent plane may get weighted using the distance to the occluder, causing underestimated occlusion, which is most visible in area *c* when $sin(t) = -1.0$.

While fixing the tangent (strategy 3) does not result in geometrically correct calculations, or even the smallest absolute out of the three options, we consider its error most suitable to the approximated nature of SSAO. Also, the aesthetically chosen falloff term can be used to compensate for underocclusion. In the following sections we have chosen to fix all tangents to $sin(t) = -0.85$ which produces mainly underocclusion.

4.2. Algorithm

Algorithm 1 lists the pseudocode for processing one line in the framebuffer. For a line of M steps there are exactly M pushes to the stack and therefore at most M pops. In addition, there is one iteration per step that terminates the loop at lines 12 – 18 without causing a pop, which results in finding the point that casts the highest obscurance. Therefore the inner loop will perform between M and $2M$ iterations in total and yields the time complexity of $O(M)$ for the algorithm.

5. Gathering Line Sweep Results

In this section we cover how lines are spread out in the framebuffer, how sampling coordinates are selected along the lines, and finally how the obscurance results from processed lines are gathered per final rendered pixel.

5.1. Line Placement

In order to densely evaluate obscurance for each of the K directions lines can be placed 1 pixel width apart and steps along each line can be chosen to be 1 pixel width long. As a result obscurance is evaluated at $W \cdot H$ pixels for each K directions (previously illustrated in Figure 4). Since the same set of azimuthal directions is shared by all pixels, banding may become visible unless a large K is used. Increasing K eventually hides banding, but render times also increase linearly in K. In order to speed up rendering when a large K is used it is often desirable to evaluate obscurance more sparsely.

Instead of placing lines 1 pixel width apart, we can spread them D_L pixel widths apart where D_L is a given fixed value. Also, instead of taking 1 pixel width steps along each line, we can take D_S pixel width steps. Having D_L and D_S larger than 1 effectively causes obscurance in one azimuthal direction to be evaluated sparsely. In order to gather the sparse results for each receiver pixel in the frame buffer, we select

Algorithm 1 SweepLine(float2 pos, float2 dir, int steps)

Functions peek1() and peek2() return the last and the second to last element of the stack, respectively.

```
while (steps−−)
{
    float3 p = deProj(sampleDepth(pos))
    float2 $p_k$ = float2(p.xy · dir, p.z)

    // Unit vector towards the camera
    float2 $u_k$ = $-p_k/||p_k||$

    float2 h1 = hull.peek1() − $p_k$
    float2 h2 = hull.peek2() − $p_k$

    while (occlusion(h1, $u_k$) < occlusion(h2, $u_k$) &&
            h1·$u_k$/||h1|| < h2·$u_k$/||h2||)
    {
        hull.pop()
        h1 = h2
        h2 = hull.peek2() − $p_k$
    }

    writeResult(occlusion(h1, $u_k$))
    hull.push($p_k$)
    pos += dir
}

float occlusion(float2 h, float2 u)
{
    // sin(t) = −0.85
    return sin(t) + max(0, h·u/||h||− sin(t)) · ρ(||h||)
}
```

the nearest point in each K direction at which obscurance was evaluated and average the results. Figure 8 illustrates sparse sweeps using $D_L = D_S = 2.0$. Each sweep therefore subsamples the depth field in a rotated grid pattern.

When the subsampled results from K sweeps are gathered, a different set of points will be selected for each screen pixel but the full azimuth of K directions is included and therefore no noise is produced to the image. Instead blur is produced, because the values that are averaged are sampled from the neighborhood of the receiver pixel and not exactly at it. To limit blurring, the average can be taken only from points that have the normal, the depth (used in this paper), or both within a threshold of the receiver, which is similar to edge-awareness used by blur filters in previous methods. Previous methods usually apply a large blur kernel to hide banding or noise in post-process, whereas we hide banding by increasing K and counter increase in execution time, in exchange for blur, by gathering sparsely evaluated obscurance values. The K nearest obscurance values for each screen pixel are found within the radius of $\sqrt{(D_S/2)^2 + (D_L/2)^2}$ which for

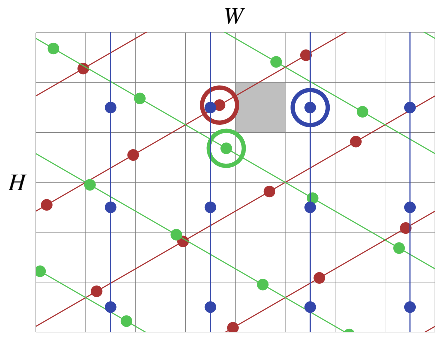


Figure 8: *Sparse ($D_S = D_L = 2$) sampling in 3 azimuthal directions ($K = 3$). The K highlighted points are selected for the screen pixel marked in gray.*

sparsities used in this paper are small compared to a typical post-process blur filter radius.

5.2. Sampling Coordinates

If the depth field is assumed not to be an infinitely thick volume, depth samples cannot be interpolated across depth discontinuities without causing temporal and spatial artefacts. In [BSD08] this is addressed by snapping sampling coordinates to texel centers, and methods relying on mipmaps (such as [MML12]) do not actually average values but instead use the values found in the base level of the depth buffer.

However, sampling coordinates along arbitrary lines in our method do not usually hit texel centers. Always snapping to texel centers, on the other hand, produces artefacts because the pattern of visited samples is not the same for every receiver along the line as shown in Figure 9. This is especially prominent on steep surfaces where a small deviation from the center of the sweep line causes large jumps in the sampled depth values. One possibility is to use linear interpolation when traversing along a line until an edge is detected (depth or normal changes too much from the previous point), in which case the sampling coordinate is snapped to texel center and sampled again. While this eliminates the sampling artefacts, it incurs some overhead and impacts performance.

However, it is possible to choose the K directions and D_S such that the snapped sampling coordinates of previously visited samples at any receiver form the same pattern. The 4 trivial cases are the axis aligned directions, for which any D_S can be used. For larger values of K, directions can be chosen using a grid of $(2n+1) \times (2n+1), n \in \mathbb{Z}^+$ texels as a template as shown in Figure 10. This gives $K = 8n$ aligned directions. For each direction we choose the step length D_S and the direction such that they match the vector from the center of the grid to each of the outer edge texel centers.

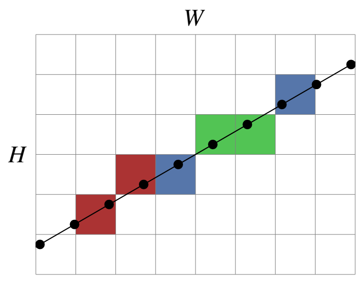


Figure 9: *The colored texels will be sampled when sampling coordinates are snapped to texel centers. For two different receivers along the line (in blue), the sampling patterns (in red and green) relative to the receiver differ (1 left/0 down and 2 left/1 down vs. 1 left/1 down and 2 left/1 down) and show up as noise in the obscurance on slanted surfaces.*

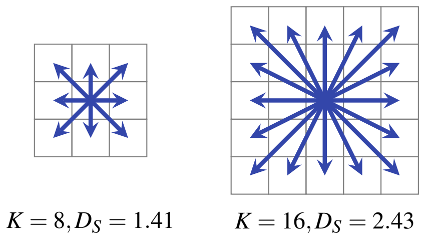


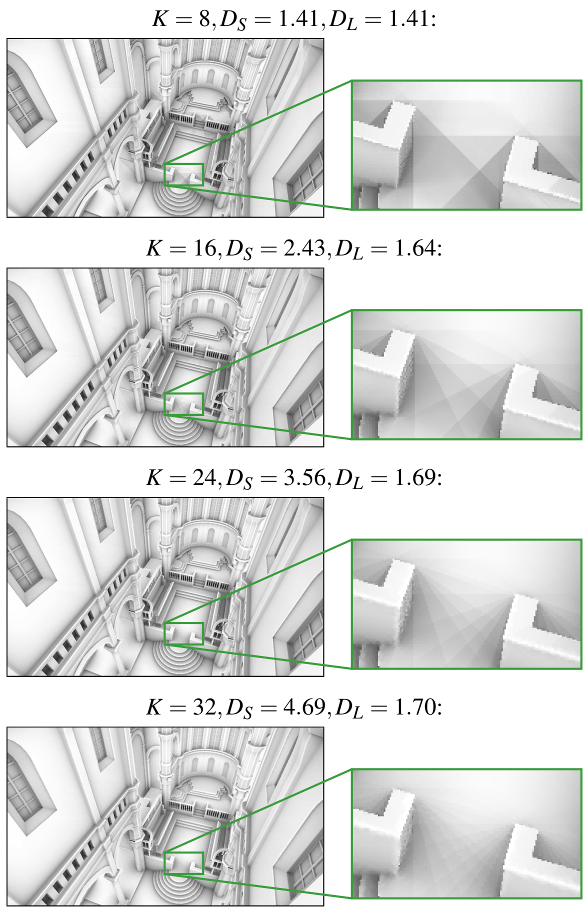
Figure 10: *Aligned sampling patterns and their average step length D_S. The axis aligned directions use D_S that is the average of the rest of the directions.*

While this allows safe snapping of sampling coordinates to texel centers, the average D_S increases along with K. Fortunately this is not a problem, since increasing K is often offset by making the sampling sparser (larger D_S and D_L) such that banding is traded for blur without increasing the execution time. The average number of calculated obscurance values per pixel is $K/(D_S \cdot D_L)$.

Choosing sampling directions according to the box pattern causes directions near the diagonals to be sampled more densely. In order to avoid bias resulting from this, contribution along directions should be weighted according to the azimuthal coverage of each direction, such that smaller weights are given to the diagonal directions. This will result in slightly higher resolution sampling near the diagonals instead of bias.

Figure 11 shows a scene with a slowly decaying falloff (to accentuate banding) and different values of K, D_S, and D_L such that the number of obscurance values per pixel remains constant. Obscurance values, here, are gathered per-pixel by respecting depth edges but ignoring normals. Respecting normals as well can be used to further contain blur.

Our implementation uses an intermediate floating point



$K = 8, D_S = 1.41, D_L = 1.41$:
$K = 16, D_S = 2.43, D_L = 1.64$:
$K = 24, D_S = 3.56, D_L = 1.69$:
$K = 32, D_S = 4.69, D_L = 1.70$:

Figure 11: *Different number of azimuthal directions K with sampling sparsities such that the density of obscurance values per pixel remains constant ($K/(D_S \cdot D_L) = 4$).*

buffer the size of $K \cdot W \cdot H/(D_S \cdot D_L)$ elements to store the obscurance values of the line sweeps. For $D_S = 4.87, D_L = 2.46, K = 16, W = 1920, H = 1080$ (Figure 13) and $D_S = 2.83, D_L = 2.12, K = 8, W = 1920, H = 1080$ (Figure 1) this is approximately 11 MB. Under tight memory constraints it is also possible to scatter the results to the framebuffer directly using atomic additions during line sweeps, but as this produces highly uncoalesced writes on a GPU we found its performance notably worse than using the intermediate buffers.

6. Results

Since traversing lines that vary in their length and thus write a different number of output values does not map well to fragment shaders, our algorithm requires either compute shaders or GPGPU. We have selected to use CUDA and made the sources available under the BSD license at http://wili.cc/research/lsao/. The benchmarks are performed on an NVidia GeForce GTX 480 GPU. The HBAO method used as the reference is implemented as an OpenGL fragment shader. For quality and performance comparison between HBAO and other recent SSAO methods, refer to [VPG13] and [McG10].

HBAO requires a falloff function that decays to 0, and we have chosen the following falloff function for HBAO:

$$\rho_0(d) = max\left(0, \frac{r(1+C)}{r+d^2} - C\right) \quad (3)$$

This function has roughly the same shape as Eqn. 2 for small C. Compared to Eqn. 2 it is sunken by C, clipped to 0, scaled to start from 1, and reaches zero at $d = \sqrt{r/C}$. In this section we use $C = 0.3$.

Figure 12 shows two scenes rendered at 1280(+256)×720(+144) (20% guard band) using two different rates of decay r for the falloff function. Our method uses configurations for $K = 8$ and $K = 16$ shown in Figure 11, whereas the number of HBAO steps N have been hand-picked for each scene and are scaled per-pixel to cover the eye-space falloff radius. The execution time of HBAO is different for the two falloff decay rates because a different number of steps has to be taken to cover the bulk of the falloff function with the same granularity. The execution time of our method depends mainly on the variance in the number of iterations of the inner loop in Algorithm 1 within warps of threads, and does not vary significantly. In all scenes our method performs roughly $2K$ iterations per pixel on average ($\approx K$ during line sweeps and K for gathering the results) whereas HBAO has to perform an order of magnitude more, $K \cdot N$.

Table 1 shows scaling with respect to screen resolution in the Sponza scene at $K = 16$ (bottom left in Figure 12). HBAO has to use larger N at higher resolutions to cover the eye-space falloff at the same screen-space accuracy, whereas our method has a constant per-pixel cost and scales linearly in the resolution. The execution time of HBAO in fact increases slightly faster than cubicly in the number of screen pixels because of increased texture cache misses, whereas the slower than quadratic scaling in the execution time of our method at lower screen resolutions is due to the small number of threads (i.e. lines to sweep) which impacts hardware utilization. Our method takes very few texture samples and is not much impacted by a texture cache miss penalty.

Table 1: *Total render times of our method and HBAO at different resolutions using 20% guard band. The scene is shown in Figure 12 to the bottom left.*

Screen resolution	Our method	HBAO
800×600	1.49 ms	10.5 ms
1280×720	2.56 ms	24.2 ms
1920×1080	5.24 ms	92.5 ms
2560×1600	9.58 ms	249 ms

Our method consists of two stages: The line-sweep stage and the result gathering stage. Table 2 shows execution time breakdown for these two stages when K is increased but $K/(D_S \cdot D_L)$ is kept constant. Even though the amount of intermediate sweep data stays the same, more data is read per pixel which shows up as a steady increase in the execution time of the gather stage. Execution time of the line-sweep stage, on the other hand, decreases slightly because the work is split into a larger number of threads that run shorter, which improves hardware utilization.

Table 2: *Render time breakdown per stage for our method at 1280(+256)×720(+144) for cases shown in Figure 11.*

Configuration	Line-sweep	Gather
$K = 8, D_S = 1.41, D_L = 1.41$	1.91 ms	0.38 ms
$K = 16, D_S = 2.43, D_L = 1.64$	1.80 ms	0.59 ms
$K = 24, D_S = 3.56, D_L = 1.69$	1.67 ms	0.73 ms
$K = 32, D_S = 4.69, D_L = 1.70$	1.60 ms	0.90 ms

Finally, two more scenes rendered at 1080p resolution are shown in Figures 1 and 13. Both scenes are rendered at 1.33 obscurance evaluations per pixel in roughly 2 ms; Figure 1 with $K = 8, D_S = 2.83, D_L = 2.12$ and Figure 13 with $K = 16, D_S = 4.87, D_L = 2.46$. Since our method uses a globally fixed set of K azimuthal directions, a small K can result in severe banding. Our primary way of fighting banding is by increasing K, and our primary way of fighting high render times due to a high K is by controlling the sparsity parameters D_L and D_S. In summary, the configuration of our algorithm is a balancing act of performance, banding, and blur: A small K and high D_L and D_S improve render times, whereas a high K reduces banding and small D_L and D_S reduce blurring.

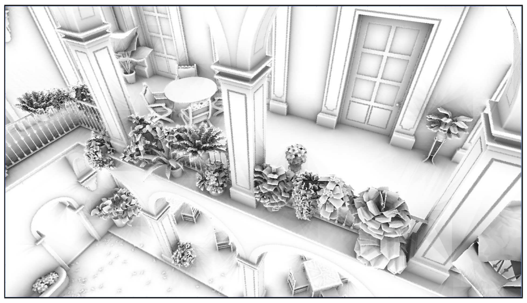


Figure 13: *SSAO rendered by our method at 1920(+192)×1080(+108) in 2.3 ms on a GeForce GTX 480.*

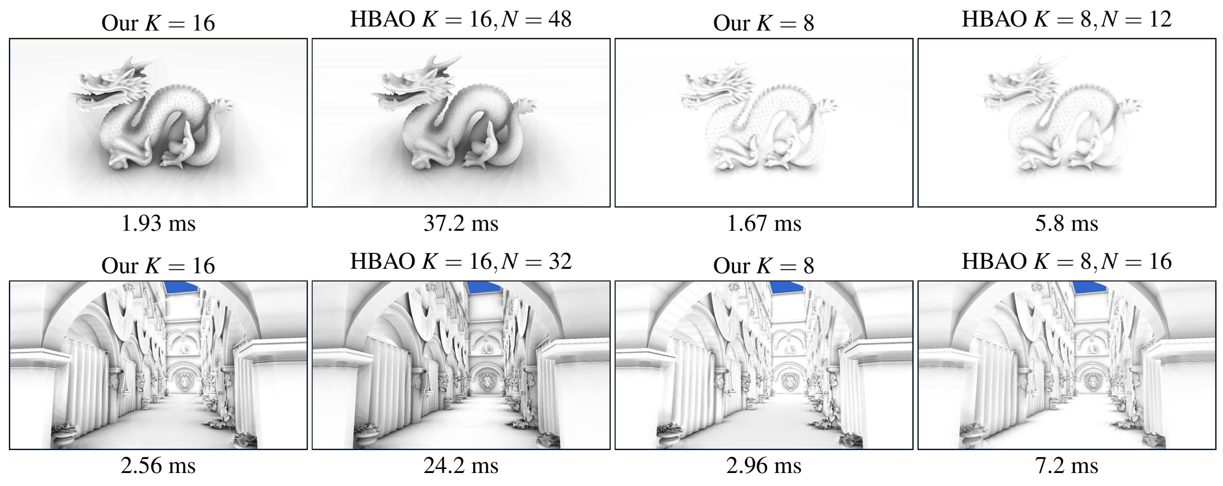


Figure 12: *Scenes rendered at 1280(+256)×720(+144) using different falloff decay rates by our method and HBAO. For HBAO, the number of steps along each of the K azimuthal directions is denoted by N.*

7. Conclusion

Our method is the first attempt to reduce the underlying time complexity of SSAO since its introduction in 2007. Previous methods rely exclusively on strategies that, in order to determine visibility of (and eventually occlusion from) m sampled scene points around n receivers, require $O(mn)$ work. Many of the m points are not visible to the receiver or cast only a small amount of occlusion. In contrast, our method finds the largest falloff weighted occluders along K azimuthal directions for n receivers in $O(Kn)$ time. The falloff radius has no impact on the performance or on the image quality of our method. The largest occluder is found in constant time at per-pixel accuracy regardless of its distance from the receiver, therefore avoiding exhaustive sampling based searches used by previous methods.

Our method uses a globally fixed set of K azimuthal directions and is prone to exhibit banding for small K. In order to avoid having to increase execution time linearly in K to hide banding, it is possible to accept blur instead by evaluating obscurance at a lower than per-pixel density along the depth field and gather, per final screen pixel, the nearest value from each azimuthal direction. Overall our method greatly improves the render times of medium to large range SSAO effects and scales well to high resolutions.

References

[Bav11] Bavoil L.: Horizon-based ambient occlusion using compute shaders. *NVIDIA Graphics SDK 11 Direct3D* (2011). 2

[BS09] Bavoil L., Sainz M.: Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH '09 Talks* (2009), ACM. 2

[BSD08] Bavoil L., Sainz M., Dimitrov R.: Image-space horizon-based ambient occlusion. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks* (New York, NY, USA, 2008), ACM, pp. 1–1. 2, 6

[FM08] Filion D., McNaughton R.: Effects & techniques. In *ACM SIGGRAPH 2008 Games* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 133–164. 3

[LS10] Loos B. J., Sloan P.-P.: Volumetric obscurance. In *Proceedings of I3D 2010* (2010), ACM. 2

[Max88] Max N.: Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer 4*, 2 (Mar. 1988), 109–117. 2

[McG10] McGuire M.: Ambient occlusion volumes. In *Proceedings of High Performance Graphics 2010* (June 2010). 8

[Mit07] Mittring M.: Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses* (2007), ACM, pp. 97–121. 2

[MML12] McGuire M., Mara M., Luebke D.: Scalable ambient obscurance. In *High-Performance Graphics 2012* (June 2012). 2, 6

[MOBH11] McGuire M., Osman B., Bukowski M., Hennessy P.: The alchemy screen-space ambient obscurance algorithm. In *Proc. HPG* (2011), HPG '11, ACM, pp. 25–32. 2

[SA07] Shanmugam P., Arikan O.: Hardware accelerated ambient occlusion techniques on gpus. In *Proc. I3D '07* (2007), ACM. 2

[TW10] Timonen V., Westerholm J.: Scalable Height Field Self-Shadowing. *Computer Graphics Forum (Proceedings of Eurographics 2010) 29*, 2 (May 2010), 723–731. 2, 4

[VPG13] Vardis K., Papaioannou G., Gaitatzes A.: Multi-view ambient occlusion with importance sampling. In *Proc. i3D* (2013), I3D '13, pp. 111–118. 8

[ZIK98] Zhukov S., Inoes A., Kronin G.: An Ambient Light Illumination Model. In *Rendering Techniques '98* (1998), Drettakis G., Max N., (Eds.), Eurographics, Springer-Verlag Wien New York, pp. 45–56. 2