6.1 Screen-Space Ambient Occlusion

VLADIMIR KAJALIN

INTRODUCTION

Ambient occlusion (AO) is a well-known computer graphics method for approximating global illumination. It is an important instrument to improve the perception of scene topology for the human eye [Luft06]. AO provides a good ratio between quality and speed, but it was never feasible to run it completely in real-time for complex and completely dynamic 3D environments in time-critical applications such as computer games, where it needs to be computed per frame within a few milliseconds.

This article will describe the algorithm and implementation of the approach that was successfully used in the PC game *Crysis*, released in 2007 (see Figure 6.1.1).

FIGURE 6.1.1 Visualization of screen-space AO component of a typical game scene in *Crysis*.

THE PROBLEMS

Most of the existing AO methods suffer from the following problems.

- **Heavy preprocessing.** Most techniques precompute AO information in a preprocess on a per-object or per-level basis. This means level designers or artists often have to wait several minutes or hours before they can see their work in a final game environment.

- **Dependency on scene complexity.** In most cases, complex and detailed scenes require much more processing time, higher memory consumption, and additional programming efforts.

- **Inconsistency between static and dynamic geometry processing.** A common way of handling AO is precomputing high-quality AO for static geometry to store it in textures and to use fast but low-quality real-time solutions for dynamic geometry. This leads to visual inconsistency between the static and dynamic components of a scene. Some of these problems can be tackled by more complex code, but complexity here is quickly exceeding the benefits.

- **Complications in implementation.** Developing a good AO compiler featuring an efficient ray-tracer, a good texture packer, and other components that can handle huge game levels full of various types of objects costs a lot of time and programming efforts. At runtime the precomputed data needs to be streamed from media and processed by the engine. Here, it competes with other systems on valuable resources such as memory, CPU, and IO bandwidth.

Our approach allows a simple and efficient approximation of AO effects entirely in real-time, free from all the problems listed above. The algorithm is executed solely on the GPU and implemented as a simple pixel shader, analyzing the scene depth buffer and extracting the occlusion factor for every pixel on the screen.

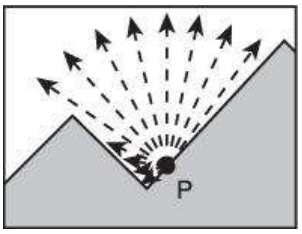
PREVIOUS WORK

The idea of extracting ambient occlusion from the scene depth buffer is not new, but it has not received enough attention in the past. The solution described in [Shanmugam07] handles near and high-frequency local ambient occlusion using a screen-space approach. More distant parts of the scene are handled using a low-detailed approximation of the scene geometry, where each object is represented as a sphere. In comparison to [Shanmugam07], our approach handles an entire scene using a single homogenous solution and is independent of scene complexity.

Overview of the Approach

Let's see how a screen-space solution can work in the simplest case. For every pixel P of the screen, we would like to compute the occlusion factor due to the surrounding geometry. Similar to most offline AO compilers, we trace many 3D rays from point P into the surrounding space and check how many of these rays hit geometry as shown in Figure 6.1.2. In order to do that on a GPU, we need a texture containing information about the scene geometry. Since we want this to be fully dynamic without any preprocess, the scene depth target seems a good candidate for the task, as the depth buffer provides a discrete sampling of the (visible) surfaces. In our renderer we had this information already available because we exported the z values to a texture in an early z pass. What problems might we encounter with such a simple brute force approach, and how we can solve them?

FIGURE 6.1.2 Typical way of computing the AO value for some point P. Black rays are hitting the geometry surface. Red rays don't hit anything.



- **Geometry outside of the camera frustum does not cast occlusion**. Information about occluders outside the camera view is missing, which leads to visual artifacts such as no darkening at the edges of the screen. This is true, but only to a certain degree. In most cases, depth values of a scene behind the edges of the screen are very similar to the values right at the border of the screen. This property of scene depth allows us to solve at least half of the cases that initially looked problematic. Another factor that helps us is the fact that human vision may not recognize small high-frequency details at long distance and at the same time may not recognize big low-frequency occlusion at very close proximity. Basically, the size of the most important details is fixed in screen-space. This leads to the conclusion that the size of an area taken into account for occlusion may be relatively small and fixed in screen-space. In practice, values of about 10% of the screen size work well. Such a solution provides a general method of computing AO for any pixel on the screen. It may handle nicely, for example, AO of a human face right in front of the camera, and at the same time huge objects such as mountains or buildings in the background (see Figure 6.1.1).

- **Geometry occluded by other geometry does not cast occlusion.** A single depth buffer cannot provide a full description of the scene topology, and information about some hidden parts of the scene may be missing. This statement is true, but in reality this problem causes few problems. Several complex solutions may be developed in order to overcome this issue, but we think artifacts that are unnoticeable to the average user are acceptable, especially because we aim for an efficient real-time solution. In the implementation described, only the foremost pixels, or basically only visible pixels, are taken into account for occlusion.

- **It will require a lot of ray tracing for every pixel on the screen.** In the case of a brute force solution for good visual quality, hundreds of rays have to be traced for every pixel on the screen, which is not affordable on current hardware. Below we describe several approximations and optimizations to allow good visual quality using a minimal number of texture reads.

- **No unified solution for alpha-blended surfaces.** As the technique depends on existing depth values, it's problematic to support alpha-blended surfaces. At this stage there's simply no good solution for the current hardware generation. In *Crysis*, the usage

of alpha blending was limited to only particles and a few other special cases when screen-space (SS) AO was not required and not used.

Amount of Geometry

Tracing many "real" rays through the depth buffer for every screen pixel (marching along the rays requires multiple texture fetches per ray) is not efficient enough on current hardware generation, so we have to find a simple approximation of this operation. Instead of computing the amount of occlusion produced by geometry hits through ray-tracing, we approximate it by the amount of solid geometry (ratio between solid and empty space) around a point of interest.

In order to do this we sample the 3D space around point P (using a predefined kernel of offset points distributed in the surrounding sphere) and for every sample point we check whether we are inside the geometry or not (see Figure 6.1.3). This can be implemented by a simple comparison between the screen-space depth of the 3D sample point and the value stored in the depth buffer at the same screen position. If the depth value of a sample point is greater than the depth buffer value, the sample point is considered to be inside the geometry.

In the case of a flat wall in front of the camera, half of the samples will appear inside the wall and half outside (Figure 6.1.4, left) so the visibility ratio is approximately 0.5. In the corner of the room (Figure 6.1.4, center) the amount of geometry is about 0.75, and the corner appears darker.

FIGURE 6.1.3 Computing the amount of solid geometry around a point P. The ratio between solid and empty space approximates the amount of occlusion.

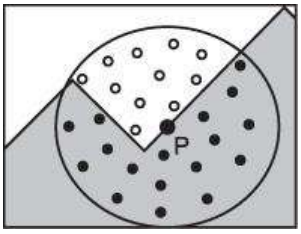
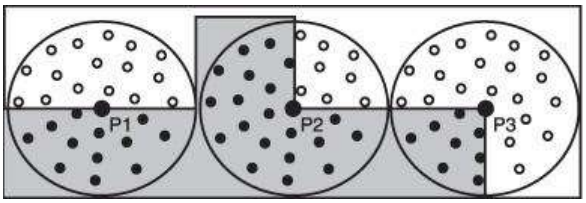


FIGURE 6.1.4 Several scenarios of AO computation. P1 is located on the flat plane, P2 in the corner, and P3 on the edge.



An interesting situation occurs, for example, on the edges of the box (Figure 6.1.4, right). The amount of geometry is about 0.25, and pixels on the edge appear brighter than on the rest of the object. At first sight such behavior may be considered a visual artifact (since in real-life the surface normal is taken into account and edges do not appearing brighter), but we found that by not fixing this "artifact" we can make more geometry features recognizable to the viewer and improve the overall 3D perception of the scene geometry (Figure 6.1.5).

FIGURE 6.1.5 Example showing the benefits from edges highlighting.

Distance Attenuation

Special computations of the distance attenuation for individual sample points are not necessary in our implementation because we solve it using a nonuniform way of distributing offset points in the sampling kernel. The density of sample points near the center of the kernel is higher than near the surface of the sphere. This way, the amount of occlusion from far geometry is weighted less than from near geometry. Additionally, since the kernel sphere is projected to a 2D screen, the density of samples near point P in screen-space is higher than on the projected sphere's boundaries. Such a distribution of sample points makes occlusion from near geometry stronger and more precise. This way, after summing up all accessibility values from all sample points, we get an overall accessibility for a point P, taking into account distance attenuation.

Depth Range Check

In some cases our method will fail, causing false occlusion from near objects to far objects in the background. This occurs when information about correct occluders is missing in the scene depth buffer. In order to compensate, an additional depth range check is used. Occlusion from sample points having a too big depth difference compared to the center point P is faded out or ignored completely and replaced with a fixed default value.

Randomly Rotated Kernel and High-frequency Noise

Since we are talking about usage of AO in time-critical applications such as computer games, we have to limit the number of texture fetches to a small number. We used 16 fetches per pixel. Such a small amount of samples would result in an image like Figure 6.1.6, which quality-wise is clearly not good enough to be acceptable.

FIGURE 6.1.6 SSAO computed without random rotation of the sampling kernel.

A common solution to improve quality with a small sample count is to use a randomly rotated kernel of sample vectors [Isidoro06]. This way we get a virtually much higher sample count, and the overall quality will improve. Unfortunately, we also introduce a lot of noise in the final picture. A common method to reduce the noise is to blur the image. Since we don't want to loose details, we can not use a big blur radius. We pick a 4×4-pixel blurring area size, which means we will be able to completely remove very-high-frequency components of the noise only in the image. In order to produce only high-frequency noise during the depth buffer sampling stage, we repeat orientations of our kernel every 4×4 screen pixels. This means that in the case of a flat wall in front of the camera, we will get a screen filled by the same tiled 4×4 pixels pattern, and then using a 4×4 post-blur step (with equal weighs for each of the 4×4 pixels), we will be able to completely remove noise (see Figure 6.1.7).

FIGURE 6.1.7 On the left, the post-blur step is disabled. On the right, post-blur is enabled.

Since each pixel in our 4×4 pattern gets a unique orientation of the kernel, we effectively simulate 256 unique vectors for every 4×4-pixel block, allowing good visual quality.

Edge-Preserving Blur

A simple blur may introduce some leaks between dark and bright areas of the AO image. For example, incorrect brightness may be noticeable on the silhouette edges of a dark object on a bright background. In order to avoid leaking, we use a more advanced version of the blur, taking into account depth discontinuities and blending only pixels with a similar depth. Other techniques used interleaved sampling strategies before, and they also blended the samples in the final pass [Wald02] [Segovia06].

Implementation

The SSAO technique requires a scene depth buffer that is available to the pixel shader. This requirement doesn't seem to be a problem since at the time we developed the technique, most graphics engines already did a z pre-pass, and some already stored the scene depth in a texture for later use. A z pre-pass allows reduction of the overdraw during consequent passes, and a scene depth texture can be used in many advanced effects such as soft particles, depth of field, atmospheric effects, deferred lighting, and shadow mapping [Wenzel07] [Mittring07].

We can run the SSAO technique after the scene depth texture becomes available. We draw a full-screen quad into the AO render target using the SSAO shader. After that, by rendering another full-screen quad, we blur the AO values with the blur method described above. The output render target will contain AO values that may be used during light passes. Usually, the AO value modulates ambient lighting, but for some stylized looks it can be even applied to diffuse and specular components.

SSAO Pixel Shader

Here we present an example of the pixel shader function computing SSAO. It is not the fastest implementation since the main purpose of this example is to help the reader understand the algorithm. This code will compile into more than 200 instructions. An optimized and vectorized but much less readable version of this code was used in *Crysis*, where it was compiled into about 86 instructions.

```
float ComputeSSAO(

  // 4×4 texture containing 16 random vectors

  sampler2D sRotSampler4x4,

  // scene depth target containing normalized

  // from 0 to 1 linear depth

  sampler2D sSceneDepthSampler,

  // texture coordinates of current pixel

  float2 screenTC,

  // dimensions of the screen

  float2 screenSize,

  // far clipping plane distance in meters
```

```
  float farClipDist)

{
// get rotation vector, rotation is tiled every 4 screen pixels

float2 rotationTC = screenTC * screenSize / 4;

float3 vRotation = 2*tex2D(sRotSampler4x4, rotationTC).rgb-1;



// create rotation matrix from rotation vector

float3x3 rotMat;

float h = 1 / (1 + vRotation.z);

rotMat._m00= h*vRotation.y*vRotation.y+vRotation.z;

rotMat._m01=-h*vRotation.y*vRotation.x;

rotMat._m02=-vRotation.x;

rotMat._m10=-h*vRotation.y*vRotation.x;

rotMat._m11= h*vRotation.x*vRotation.x+vRotation.z;

rotMat._m12=-vRotation.y; rotMat._m20= vRotation.x;

rotMat._m21= vRotation.y; rotMat._m22= vRotation.z;



// get depth of current pixel and convert into meters

float fSceneDepthP = tex2D( sSceneDepthSampler, screenTC ).r *

                     farClipDist;



// parameters affecting offset points number and distribution

const int nSamplesNum = 16; // may be 8, 16 or 24

float offsetScale = 0.01;

const float offsetScaleStep = 1 + 2.4/nSamplesNum;



float Accessibility = 0;



// sample area and accumulate accessibility

for(int i=0; i<(nSamplesNum/8); i++)

for(int x=-1; x<=1; x+=2)

for(int y=-1; y<=1; y+=2)

for(int z=-1; z<=1; z+=2) {

  // generate offset vector (this code line is executed only

  // at shader compile stage)
  // here we use cube corners and give it different lengths

  float3 vOffset = normalize(float3( x, y, z )) *

                   ( offsetScale *= offsetScaleStep );



  // rotate offset vector by rotation matrix

  float3 vRotatedOffset = mul( vOffset, rotMat );



  // get center pixel 3d coordinates in screen space

  float3 vSamplePos = float3( screenTC, fSceneDepthP );
```

```
  // shift coordinates by offset vector (range convert
  // and width depth value)
  vSamplePos += float3( vRotatedOffset.xy,
                        vRotatedOffset.z * fSceneDepthP * 2);

  // read scene depth at sampling point and convert into meters
  float fSceneDepthS = tex2D( sSceneDepthSampler,
                       vSamplePos.xy ) * farClipDist;

  // check if depths of both pixels are close enough and
  // sampling point should affect our center pixel
  float fRangeIsInvalid = saturate( ( ( fSceneDepthP -
                          fSceneDepthS ) / fSceneDepthS ) );

  // accumulate accessibility, use default value of 0.5
  // if right computations are not possible
  Accessibility += lerp( fSceneDepthS > vSamplePos.z, 0.5,
                   fRangeIsInvalid );
}

// get average value
Accessibility = Accessibility / nSamplesNum;
  // amplify and saturate if necessary
  return saturate( Accessibility*Accessibility + Accessibility );
}
```

Future Improvements

SSAO is a new technique and has a lot of untapped potential for quality and speed improvements, and we cover only few of them here.

For every pixel on the screen our shader does many reads from a relatively large-sized texture at completely different positions, causing heavy texture cache trashing. Even code vectorization and reducing the shader instruction count two to three times wasn't increasing speed because of the texture lookup performance. One way to get better performance is to use a downscaled version of depth texture. In *Crysis* this optimization allowed us to roughly double the speed of the SSAO shader. Another possible example of code optimization is to replace the rotation of offset vectors using matrix operations with faster mirroring of the vector by plane. The mirror trick has been used in *Crysis* and is described in [Mittring07].

An interesting possibility would be to take scene normals into account. One of the obvious benefits from it can be to use a normal in order to concentrate more samples in front of the face and avoid wasting half of the samples for self-occlusion.

The SSAO technique is an approximation of the ambient occlusion, which itself is an approximation of the indirect lighting. The SSAO technique can be extended to approximate indirect lighting even better by taking into account colors of the scene. Such a screen-space indirect lighting solution (SSIL would be a good name) result is a more plausible approximation of indirect lighting taking into account brightness and colors of surfaces. An image-space indirect lighting technique has been described by [Dachsbacher05].

Results and Conclusion

This article described the implementation of a technique for computing AO using only the scene depth buffer. Several optimizations and approximations were developed in order to speed up the process while maintaining high visual quality. Performance-wise, the implementation presented here requires 3 milliseconds per frame on an NVIDIA GeForce 8800

GTX at 1280×720 screen resolution, including the aforementioned post-blur step. Such a good performance makes it finally possible to use real-time AO in time-critical applications such as computer games.

During the development of *Crysis*, after researching multiple AO solutions, we found that SSAO is a very effective solution for us. Major benefits we found are improved game production, support for dynamic content, simple implementation, and adjustable quality. As a disadvantage we can mention the requirement for a good graphic card like the GeForce 8800, which was relatively expensive at the time of the *Crysis* release. Nowadays such hardware is a usual component in a gamer PC, and soon nothing will prevent SSAO from running on most computers or game consoles; we expect that in the near future all graphics engines will support SSAO (or SSIL) as a common feature.

Acknowledgments

Special thanks to the Crytek team for helping me develop this technique.

References

[Dachsbacher05] Carsten Dachsbacher, Marc Stamminger, "Reflective Shadow Maps," Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games

[Isidoro06] John R. Isidoro, "Shadow Mapping: GPU-based Tips and Techniques," GDC 2006

[Luft06] Thomas Luft, Carsten Colditz, Oliver Deussen, "Image Enhancement by Unsharp Masking the Depth Buffer," International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), 2006

[Mittring07] Martin Mittring, "Finding Next Gen - CryEngine 2," Advanced Real-Time Rendering in 3D Graphics and Games Course, SIGGRAPH 2007

[Segovia06] Benjamin Segovia, Jean-Claude Iehl, Richard Mitanchey, Bernard Péroche, "Non-Interleaved Deferred Shading of Interleaved Sample Patterns," Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2006

[Shanmugam07] Perumaal Shanmugam, Okan Arikan, "Hardware Accelerated Ambient Occlusion Techniques on GPUs," Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games

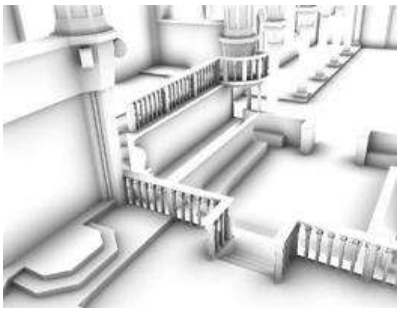
[Wald02] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, Philipp Slusallek, "Interactive Global Illumination Using Fast Ray Tracing," Proceedings of the 13th EUROGRAPHICS Workshop on Rendering, 2002

[Wenzel07] Carsten Wenzel, "Real-Time Atmospheric Effects in Games Revisited," Conference Session GDC 2007

6.2 Image-Space Horizon-Based Ambient Occlusion

LOUIS BAVOIL AND MIGUEL SAINZ

FIGURE 6.2.1 Image rendered with our horizon-based algorithm.



INTRODUCTION

AMBIENT OCCLUSION

Ambient occlusion is a lighting model that approximates the amount of light reaching a point on a diffuse surface based on its directly visible occluders. It gives perceptual clues of depth, curvature, and spatial proximity. The term *ambient occlusion* was introduced by [Landis02] and more precisely defined in [Christensen03]. [Langer99] used the same model for approximating the lighting on a cloudy day, and [Zhukov98] presented a similar lighting model called obscurances.

One way of rendering ambient occlusion is to trace rays from a surface point P in the hemisphere oriented around the normal **n** at P, and count how many of these rays intersect any surrounding object within a given radius of influence R, normalizing the result by the number of rays. The radius of influence R is important to control the set of potential occluders.

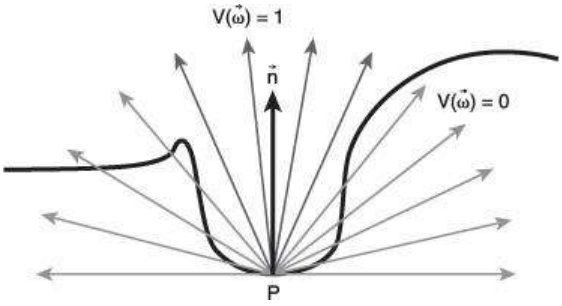
A first form of ambient occlusion uses a cosine-weighted distribution, tracing more rays toward the normal. This form is described by the following integral defined over the normal-oriented unit hemisphere Ω at point P:

$$A_1 = 1 - \frac{1}{\pi}\int_{\Omega} V(\vec{\omega})(\vec{\omega}.\vec{n})d\omega$$

EQUATION 6.2.1

Where $\mathbf{V}(\vec{\omega})$ is the visibility function returning 1 if a ray starting from P in direction $\vec{\omega}$ intersects an occluder around P within a given radius of influence R, and 0 otherwise (Figure 6.2.2), $\mathbf{d\omega}$ and is an elementary solid angle on the hemisphere Ω.

FIGURE 6.2.2 Rays distributed in the normal-oriented hemisphere.



The $(\vec{\omega}\vec{n})$ cosine term in the ambient occlusion of the type described in Equation 6.2.1 is the response to the diffuse reflection of the ambient light on the surface. Responses to other light sources should be added to A_1 to produce the final color.

A second form of ambient occlusion used by [Pharr04] and [Hegeman06] does not include the cosine term in its definition:

$$A_2 = 1 - \frac{1}{2\pi}\int_{\Omega} V(\vec{\omega})\, d\omega$$

EQUATION 6.2.2

In this case, the ambient occlusion term A_2 can be considered as the shadow contribution from an environment light. Like for regular shadowing, it should be multiplied with the direct lighting term, as described by [Hegeman06].

We follow the latter approach and solve the following ambient occlusion integral:

$$A = 1 - \frac{1}{2\pi}\int_{\Omega} V(\vec{\omega})W(\vec{\omega})\, d\omega$$

EQUATION 6.2.3

where $\mathbf{W}(\vec{\omega})$ is an attenuation function based on the distance between P and the occluder in direction $\mathbf{V}\vec{\omega}$. The purpose of the attenuation function is to soften sharp occlusion boundaries due to occluders at a distance R strongly influencing the ambient occlusion once they begin to be sampled. See Figure 6.2.3.

FIGURE 6.2.3 Ambient occlusion rendered with our algorithm, with and without an attenuation function.

In real-time graphics, for static scenes, ambient occlusion terms can be precomputed at vertices or in light maps. The problem with most techniques is that they require scene-dependent precomputations, which makes them impractical for complex scenes with arbitrary dynamic geometry.

Screen-Space Ambient Occlusion

The term *screen-space ambient occlusion* (SSAO) was introduced by [Mittring07]. [Shanmugam07] and [Fox08] described different SSAO algorithms. The general idea is to use the depth buffer of the scene being rendered as a discrete approximation of the scene geometry. Although this approach uses a single layer of depth and no information is available outside the view frustum, it produces plausible results as seen in Figure 6.2.1.

Per-pixel ambient occlusion is computed in a postprocessing pass that samples a depth image rendered from the camera (eye's point of view). This approach requires no scene-dependent precomputations.

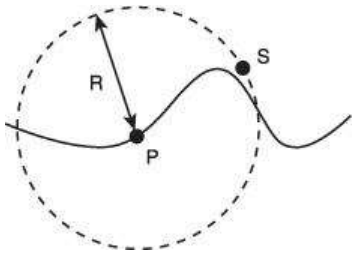
The image-based algorithm from [Shanmugam07] fits a microsphere around every pixel in the depth buffer and accumulates the ambient occlusion contribution from each microsphere occluder. The algorithm assumes that any microsphere is visible from P and does not take mutual occlusion between spheres into account. For large radii of influence, this accumulation approach produces unrealistic over-occlusion that cannot be simply fixed using a uniform scale

factor.

The crease shading algorithm from [Fox08] uses the same approach as [Shanmugam07] but without the microspheres. As it accumulates occlusion, the algorithm ignores the solid angles. This can be seen as an approximation of [Shanmugam07], where all the microspheres have the same solid angle.

The algorithm presented by [Mittring07] and [Moller08] distributes points in 3D space in a sphere around P and compares the depth of each 3D sample with the corresponding depth in the depth buffer, similar to shadow mapping. This algorithm assumes that points that are above the height field are visible from the surface point P, which is not always true (see Figure 6.2.4).

FIGURE 6.2.4 Sample S is above the height field but is not visible from P.



Sampling the Hemisphere

Equation 6.2.3 can be integrated using a Monte Carlo approach, sampling directions on the normal-oriented hemisphere and evaluating the visibility function V and the attenuation function W for each sampled direction. In offline renderers, this is typically done by analytically intersecting 3D rays with the scene geometry, using a precomputed acceleration structure to avoid testing each ray with every primitive.

Input Buffers for Image Space Ambient Occlusion

Depth Image

We store eye-space z coordinates in the depth image, and we reconstruct 3D eye-space positions from the 2D pixel coordinates and the associated depth value, assuming that the depths are being generated at the pixel centers. Now that we have this depth image available, the problem is how to integrate Equation 6.2.3 for every pixel on the screen.

Normals

As shown in Figure 6.2.2, the hemisphere is defined based on the normal direction, such that all the sampled directions have $(\vec{\omega}.\vec{n})>0$. The reason for this is that the integration domain for Equation 6.2.3 is the positive hemisphere defined by the surface normal and the point being evaluated.

In the following, we assume that we have a depth image rendered from the eye's point of view, along with the associated per-pixel normals.

Image Space Ambient Occlusion with Ray Marching

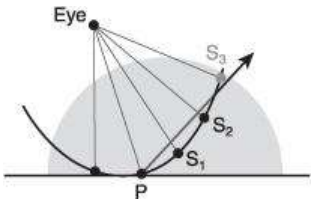
In this section, we describe a brute force algorithm for rendering ambient occlusion based on tracing rays in image space using a per-pixel depth and normal buffers. This algorithm will provide the reader a basis to understand the mechanics of the integration of the ambient occlusion in screen-space. Further in the chapter we will present our approach that follows similar underlying principles.

Intersecting a Ray With a Height Field

A simple and robust way to render ambient occlusion in image space is to consider the depth image as a height field, and intersect 3D rays with it, by stepping on the height field along a 2D ray segment and testing at each step if the ray crosses the height field surface. This same approach is also referred to as grid tracing [Musgrave88], image-based ray tracing [Lischinski98], discrete ray tracing [Yagel92], linear search [Policarpo05] [Tatarchuk06], rasterization-based intersection [Baboud06], scan-line conversion [Xie07], and 2D iterative search [Davis07]. In this article, we refer to it as ray marching, following the naming from [Perlin89].

For intersecting a ray with a height field, assuming that the ray is traced in a direction going through the normal-oriented hemisphere, we approximate the intersection point by taking the first sample for which the current position along the ray is below the height field (see Figure 6.2.5).

FIGURE 6.2.5 Intersecting a ray with a depth image by stepping along the ray and comparing the depths of the samples with the associated depths along the ray.

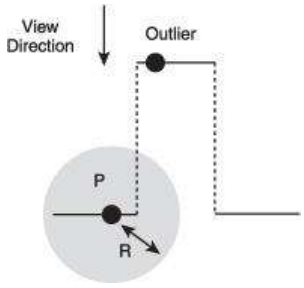


We define the end point E to be at distance R from P along the current direction. Ray-marching algorithms then step from P to E with a uniform step size. The ray marching can be done by projecting P and E first and then stepping in image space (uv = uv + step), or by stepping in 3D space (P = P + step) and projecting every sample point into image space to sample the depth image.

Avoiding Outliers

The height field that we are working on has depth discontinuities between objects (for instance, see Figure 6.2.6).

FIGURE 6.2.6 A surface point P and an outlier outside of the radius of influence R from P.



To avoid any artifacts from intersecting portions of the height field belonging to outliers (see Figure 6.2.6), we simply ignore samples that are outside the radius of influence R by computing the distance r between P and the sample position for every sample along the ray segment.

Monte Carlo Integration

If we define for each pixel a tangent-binormal-normal (TBN) basis, the normal-oriented hemisphere can be sampled using a spherical coordinate system with the zenith axis aligned with the normal at the current point P. To integrate the ambient occlusion from Equation 6.2.3 using this approach, we would need N_d directions (θ angles), N_r rays per direction (φ angles), and N_s steps per ray. When a ray intersects the depth image at point S, an ambient occlusion contribution $\mathbf{W(llP-Sll)/(N_rN_s)}$ would be added to the current ambient occlusion for surface point P. The per-pixel complexity of the algorithm is $O(N_d\ N_r\ N_s)$. For a given direction θ around

the normal, the ray-marching algorithm traces one ray per φ angle, and in practice, to get good results at least three rays per direction are required.

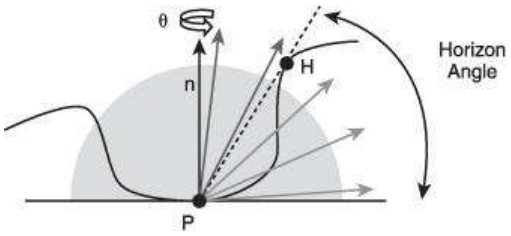
Our Algorithm

Horizon Culling

We assume that the height field from the depth image is continuous, which tends to be true in practice when ignoring outliers outside of the radius of influence from P. To simplify this explanation, we can assume that the attenuation function is $\mathbf{W(\vec{\omega})=1}$ In this case, there exists a horizon angle below which all the rays are guaranteed to intersect the height field. This is a property of height fields used for occlusion culling [Rogers85] and horizon mapping [Max86] [Sloan00].

Given a horizon point H inside the radius of influence R, the ambient occlusion contribution subtended by the horizon angle (P,T,H) and the distance ||P-H|| can be integrated analytically (see Figure 6.2.7). The problem is then how to find an accurate horizon angle in a given direction.

FIGURE 6.2.7 Horizon angle defined in tangent space for a particular direction θ. The rays with angles lower than the horizon angle intersect the depth image, and the others do not.



Reformulating the Ambient Occlusion Integral

Working in Image Space

Instead of distributing directions in eye space based on the TBN basis at the surface point P (as described in “Ray Marching” above), our new algorithm distributes directions in image space around the current pixel. This works because a property of the perspective projection is that lines in eye space project to lines in image space. Our algorithm samples depths in image space by stepping along 2D directions, and it integrates the ambient occlusion by using 3D eye-space positions.

Snapping Texture Coordinates

For each 3D sample S, to avoid any discrepancy between the fetched S.z and the exact depth associated with the offset S.xy, before sampling the depth buffer, we snap the (u,v) coordinates to pixel centers when reconstructing S.xy from (u,v). Because of this snapping requirement, we cannot use optimizations based on maximum mipmaps such as [Dachsbacher07] where the 2D locations of the depth samples in the mip levels would be unknown.

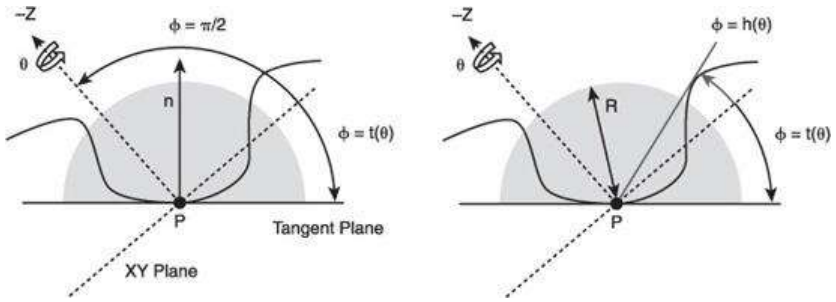
Eye-Space Basis

Our algorithm distributes rays in the hemisphere by using the eye-space XYZ basis, not the tangent-space TBN basis. We effectively rotate the TBN basis to face the camera. We are assuming a left-handed projection, so the camera is looking toward the Z > 0, parallel to the Z axis.

We parameterize the integral using spherical coordinates, with the zenith axis aligned with the Z axis in eye-space, and the azimuth angle θ around Z and the elevation angle φ. In the coordinate system, the ambient occlusion integral from Equation 6.2.3 can be expressed by Equation 6.2.4 (see Figure 6.2.8a):

$$A = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi}\int_{\phi=t(\theta)}^{t(\theta)+\pi/2} V(\vec{\omega})W(\vec{\omega})\,d\omega \qquad \text{EQUATION 6.2.4}$$

FIGURE 6.2.8 (a) Parameterization of the normal-oriented hemisphere. (b) The horizon angle h(θ).



Similarly to horizon mapping [Max86] [Sloan00], we split the unit sphere by a horizon line defined by the signed horizon angle h(θ) relative to the XY plane going through P and perpendicular to Z. In addition, we split the hemisphere by a tangent line defined by the signed angle t(θ) between the tangent plane and the XY plane (see Figure 6.2.8b).

Assuming that the neighborhood of **P** defined within the radius of influence R is a continuous height field, rays that would normally be traced below the horizon are known to intersect an occluder, that is, $\mathbf{V}(\vec{\omega})$=1 for all φ between t(θ) and h(θ). Then Equation 6.2.3 can be rewritten as

$$A = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi}\int_{\phi=t(\theta)}^{h(\theta)} W(\vec{\omega})\cos(\phi)\,d\phi\,d\theta \qquad \text{EQUATION 6.2.5}$$

We assume that a uniform distribution of directions in image space approximately corresponds to a uniform distribution of θ angles in eye space.

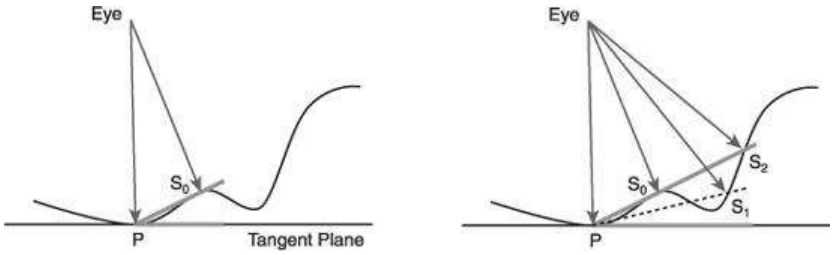
Incremental Horizon Angle

We step in the depth image in direction θ with a uniform step size and reconstruct the eye-space positions S_i from the sampled depths. The signed elevation angles $\varphi(S_i)$ are given by

$$\tan\phi(S_i) = \frac{(P - S_i).z}{\|(P - S_i).xy\|} \qquad \text{EQUATION 6.2.6}$$

In this sampling process, we ignore samples that are outside the radius of influence R, that is, $\|\mathbf{S_i}\text{-}\mathbf{P}\|\mathbf{>R}$. We also keep track of the maximum elevation angle (incremental horizon angle h) as shown in Figure 6.2.9. If the elevation angle $\varphi(S_i)$ of the current sample is not greater than $\varphi(S_{i-1})$, then we ignore sample S_i and go to the next sample. Otherwise, we compute the ambient occlusion contributed by the sample.

FIGURE 6.2.9 Horizon angles for a given direction after one sample (left) and after three samples (right).



Attenuation Function

Up to this point, we have been assuming that $\mathbf{W(\vec{\omega})=1}$ The attenuation function is important to avoid any discontinuities in the ambient occlusion and can be defined as a function of the distance to the horizon point ([Dimitrov08] [Bavoil08]). This would approximate $\mathbf{W(\vec{\omega}_i)}$ by the lowest attenuation for all samples along the direction $\vec{\omega}$, and therefore this results in over-attenuating the ambient occlusion. We found that it made a significant difference to evaluate the attenuation function $\mathbf{W(\vec{\omega}_i)}$ for every sample.

In offline rendering, the attenuation function W(r) (referred to as "falloff" in [Gritz06]) is typically set to a linear decay W(r) = (1 − r), where r is the distance to the occluder, normalized by the radius of influence R. To attenuate the samples near the middle of R less, we use the following quadratic attenuation function: W(r) = (1 − r^2), which attenuates less than the linear attenuation function.

Integrating the Ambient Occlusion

By sampling along a line on the image and skipping the samples that do not increase the horizon angle as shown on Figure 6.2.9, we get a partitioning of the elevation angles $\mathbf{\varphi\in[t(\theta),h(\theta)]}$ with $\mathbf{t(\theta)=\varphi_0\leq\varphi_1\leq\varphi_2\leq...\leq\varphi_{N_s}=h(\theta)}$, which we use to integrate Equation 6.2.5. We use a piecewise constant approximation of $\mathbf{W(\vec{\omega})}$, where $\vec{\omega}_i$ is the 3D direction associated with $\mathbf{\varphi=\varphi_i}$:

$$\int_{\phi=t(\theta)}^{h(\theta)} W(\vec{\omega})\cos\phi d\phi = \sum_{i=1}^{N_s}\int_{\phi=\phi_{i-1}}^{\phi_i} W(\vec{\omega}_i)\cos\phi d\phi$$

EQUATION 6.2.7

which yields the following incremental computation of the ambient occlusion:

$$A = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi}\sum_{i=1}^{N_s} W(\vec{\omega}_i)(\sin\phi_i - \sin\phi_{i-1})d\theta$$

EQUATION 6.2.8

To integrate Equation 6.2.8, for every 2D direction θ, we step on the depth image, and for every sample S_i, we compute $\tan(\varphi(S_i))$ and $\sin(\varphi(S_i))$. This works correctly when the samples S_i all have the same exact θ angle relative to the Z axis. However, because we need to snap the sample coordinates to the pixel centers, the θ_i angles of the snapped sample coordinates are slightly different from the θ_i angles of the true non-snapped coordinates. Taking the difference of $\sin(\varphi(S_i))$ and $\sin(\varphi(S_{i-1}))$, where S_i and S_{i-1} have perturbed θ angles, may generate objectionable false-occlusion artifacts.

To solve this issue, we compute a $\sin(t_i)$ value per sample S_i, and we rewrite the integral in terms of differences between $\sin(\varphi_i)$ and $\sin(t_i)$. This can be seen as grounding the height $\sin(\varphi_i)$ of each sample by the height $\sin(t_i)$ of its associated tangent. Equation 6.2.9 describes the grounded form of our horizon-based integral:

$$A = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi}\sum_{i=1}^{N_s} W(\vec{\omega}_i)[(\sin\phi_i - \sin t_i) - (\sin\phi_{i-1} - \sin t_{i-1})]d\theta$$

EQUATION 6.2.9

The advantage of using per-sample tangents with Equation 6.2.9 is that the integral does not produce any false-occlusion artifacts when the θ angles are perturbed by per-sample

snapping.

Implementation Considerations

Provided we have a linear depth buffer and a normal buffer, we can proceed to compute the per-pixel ambient occlusion by rendering a full-screen quad into an ambient occlusion buffer (8 bits, 1 channel). In the following subsections we present details on implementation tradeoffs in order to perform an efficient screen-space ambient occlusion integration.

Per-Pixel Normals

Normals are typically defined at vertices of a mesh and smoothed using linear interpolation during rasterization. Because per-vertex smoothed normals are by definition not orthogonal to their associated triangles, the resulting interpolated per-pixel normals will not be either, and this may generate artifacts especially visible for large triangles (low tessellated scenes). Therefore, we prefer to compute per-pixel normals in the pixel shader by taking derivates of eye-space coordinates instead of interpolated per-vertex normals.

In our implementation we have seen that 8 bits per component provides enough precision for the normal buffer, so we use a R8G8B8A8_SNORM normal texture.

Combining with the Color Buffer

There are multiple ways the ambient occlusion can be applied to a scene. For applications that use a depth pre-pass before shading, the ambient occlusion can be rendered right after the depth pre-pass and applied in a flexible manner during shading. Otherwise, the ambient occlusion should be multiplied over the current scene's color buffer right after shading the opaque objects.

We apply the ambient occlusion pass on the opaque geometry only, before any semitransparent objects are rendered over it.

Our ambient occlusion term does not include cosine terms, so it makes the most sense to consider it as an ambient shadow and apply it by multiplying the ambient terms. In our results, we simply multiply the ambient occlusion over the full shaded colors, including direct lighting terms.

Directions and Jittering

For each angle θ, we sample the depth image along a line segment in image space. The eye-space radius of influence R is projected onto the image plane, and its projected size is subdivided into N_s steps of uniform lengths.

We pick N_d 2D directions θ in image space around the current pixel, which correspond to directions around the Z axis in eye space (see Figure 6.2.10). To avoid banding artifacts, we rotate the 2D directions by a random per-pixel angle, and we jitter the samples along each direction. These randomized values are precomputed and stored in a tiled texture containing **(cos(α),sin(α),β),** where **α∈[0,2π/Nd]** and **β∈[0,1)**. The sample locations for a given direction θ are

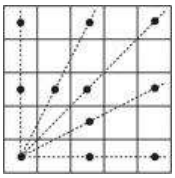
$$uv(S_i) = uv(P) + (\beta + i)(\Delta u, \Delta v)R(\alpha) \quad \text{EQUATION 6.2.10}$$

where **R(α)**is a 2×2 rotation matrix, and **(Δu,Δv)** is the step size in direction θ.

Note that this sampling distribution is biased toward P. This is not a problem, though, since our integration does not assume a uniform distribution. In fact, having more samples near the center is important for improving the quality of contact occlusion, because the nearer the samples are to the center, the less attenuated they are likely to be.

To account for the pixel-snapping requirements of the algorithm, we would be required to snap each of these **$uv(S_i)$** texture coordinates. However, it is more efficient to snap the texture coordinate of the initial sample S_0 and then snap the step size to guarantee that we always sample at pixel centers (see Figure 6.2.10). However, using such snapped directions generates banding for high numbers of steps or for small radii of influence, due to the limited number of possible directions.

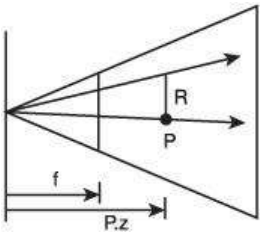
FIGURE 6.2.10 Example of snapped directions.



Variable Step Size

The step size is the distance in uv texture space between two consecutive samples along a given direction of integration. To compute it, we start by projecting the size of the radius of influence R from eye space to clip space, using similar triangles (Figure 6.2.11).

FIGURE 6.2.11 Projecting the radius of influence R into uv space.

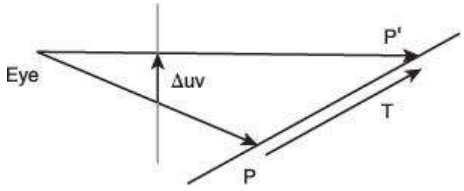


The projected size of the radius R on the focal plane is $\mathbf{R_{uv}=fR/P.z}$, where f is the focal length vector defined as (cotan(fovy/2) height/width, cotan(fovy/2)). We then compute the uv-space step size $(\Delta u, \Delta v) = (R_{uv} / 2) / (N_s + 1)$, and early exit if this step size is smaller than one texel in the *X* or the *Y* directions to guarantee, at most, one sample per texel.

Eye-Space Tangent Vectors

For a given direction θ, we start the ambient occlusion integration by computing a tangent vector T in eye space. Given a step size in texture space Δ uv associated with direction θ and given the plane equation for the tangent plane at P defined using the eye-space normal at P, the tangent vector T can be computed by intersecting a view ray with the tangent plane, as shown on Figure 6.2.12.

FIGURE 6.2.12 Tangent vector associated with a step size Δuv.



We use a simpler method to compute the tangent T using a screen-aligned basis (dPdu, dPdv) for the tangent plane, where

$$\Delta P = \Delta u \, dPdu + \Delta v \, dPdv \quad \text{EQUATION 6.2.11}$$

This means that when (u,v) moves to (u,v) + (Δu, Δv), P moves to P + ΔP, which defines a tangent vector $T(\theta) = T(\Delta u, \Delta v) = \Delta P$.

For an orthogonal projection, (u,v) is linearly related to the eye-space coordinates (x,y). However, for perspective projections, the relationship between texture-space (u,v) and eye-space (x,y) is not linear except when the plane has a constant depth. We found that the error introduced by the non-linearity does not generate any visible artifacts compared to ray casting. For efficiency, we use Equation 6.2.11 even though it is not perspective correct.

Angle Bias

When the level of tessellation of the geometry used in the scene is low, the ambient occlusion contribution tends to magnify the creases due to large polygons that produce unpleasant artifacts (see Figure 6.2.13). If increasing the polygon count in the scene is not possible, this effect can be eliminated by adding an angle bias to the horizon calculation (see Figure 6.2.14). Moreover, using an angle bias removes artifacts generated by clamping to edge when sampling depth.

FIGURE 6.2.13 (a) Low-tessellated mesh. (b) Adding an angle bias fixes the creases.

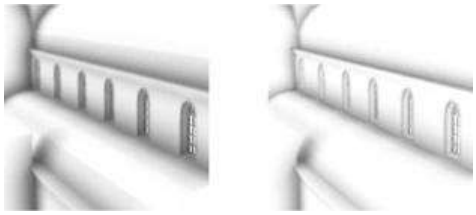
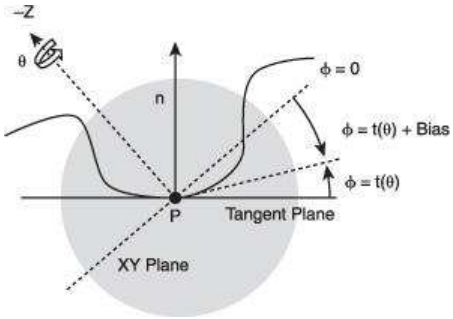


FIGURE 6.2.14 The angle bias ignores directions with tangent angles below the bias.



Cross-Bilateral Filter

Depending on the number of sampling directions and the use of jittered sampling, the result of the ambient occlusion pass presents a certain degree of noise. This can be reduced by blurring the ambient occlusion image. To preserve sharp geometric edges between foreground and background objects, we use a cross-bilateral filter based on the depth delta from the kernel center [Eisemann04].

The size of the filter kernel is a parameter. The values we found to work quite well for normal settings are between 9×9 and 21×21 pixels. As the number of directions and the number of steps increase, the blur kernel can be reduced considerably. Although bilateral filters are mathematically non-separable, similarly to [Segovia06], we have obtained good performance results by separating the filter in X and Y with very minimal visual artifacts.

Multisample Antialiasing

For integrating the ambient occlusion with multisample antialiasing (MSAA), we take as input multisample depth and normal textures generated using multiple render targets (rendering colors, depths, and normals in the same pass). First, we resolve the depth and normal textures with a full-screen pass that copies one of the samples per pixel into non-multisample depth and normal textures. Second, we perform the ambient occlusion full-screen pass, which takes as input the previously generated textures and writes into a non-multisample ambient occlusion texture. Finally, we apply our two blur passes over the ambient occlusion, taking as input the multisample depth texture and the non-multisample ambient occlusion texture.

Since the blur is performed on the non-multisample ambient occlusion texture, it tends to magnify bleeding artifacts at the edges (see Figure 6.2.15). To overcome this problem we add an edge detection pass [Cantlay07] followed by a supersampling blur pass only for the edges

that compute the blur for each depth sample per pixel, and averages the results.

FIGURE 6.2.15 (a) Original image. (b) Edge bleeding with a naïve blur. (c) Running a supersampling blur at the edges reduces the leaking and the aliasing.

Working at Lower Resolution

Since ambient occlusion is a soft global illumination effect, it is not required to support higher-frequency textures and silhouettes, and hence the calculations can be performed at a coarser level. As a performance optimization with little impact in visual quality, we propose to run the ambient occlusion pass in a half resolution buffer and up-sample the result during the edge-preserving Gaussian blur. As we will see in the experiments section, this solution is a good balance point.

Summary of the Parameters

Overall, our implementation has the following parameters that control quality and performance.

- The radius of influence R, in eye space
- The number of directions N_d distributed in image space around every pixel
- The number of steps per direction N_s (see Figure 6.2.16)
- The angle bias (see Figure 6.2.13)
- A scale factor multiplying the ambient occlusion
- The blur kernel size in pixels
- The depth variance for the cross-bilateral filter

Results

In this section, we present some results of our screen-space ambient occlusion algorithm. All the experiments have been run on an Intel Core2Duo CPU, with 2 GB of RAM, using the indicated GPUs. The data sets used for these experiments are the Stanford Dragon, the Sibenik Cathedral, and the Cornell box.

To test the performance of the algorithm, we use the Sibenik dataset with three different GeForce GPUs: 8600 GTS, 8800GT, and 8800 GTX Ultra. Table 6.2.1 contains frames per second at 1600×1200 resolution for the cases of:

- No AO/no blur: Represents the speed-of-light reference.
- No AO: Only applies the blur.
- 4,4: Uses four directions and four samples per direction.
- 8,8: Uses eight directions and eight samples per direction.
- 16,16: Uses 16 directions and 16 samples per direction.

TABLE 6.2.1 The blur size in all the results was 9×9. The half resolution AO results did not have MSAA enabled

	Frames Per Second—1600x1200				
	No AO/No Blur	No AO	4/4	8/8	16/16
	Half Resolution AO				
8600 GTS	88	43	26	15	6
8800 GT	234	127	81	48	20
8800 Ultra	301	155	100	59	24
	1x MSAA				
8600 GTS	86	42	10	4.2	1.5
8800 GT	234	128	26	10	4
8800 Ultra	299	155	41	17	5.8
	4x MSAA				
8600 GTS	38	26	8.8	4.2	1.5
8800 GT	88	67	22	9.8	3.9
8800 Ultra	138	97	36	16	5.7

In Figure 6.2.16, we show the visual quality that can be achieved with different numbers of samples. For the cases where only four directions and four steps per direction are used, we can easily obtain very visible and smooth creases. As we increase the number of total samples, more subtle details come to life, bringing more details in the scene. The main problem at low sample rate is the need for extra blur strength to compensate for the noise generated during jittering.

As for final results, we show in Figure 6.2.17 the ability of the ambient occlusion contribution to provide proper depth cues and realism to a rendered scene. We show the separate diffuse pass and the ambient occlusion pass for the dragon model and the result of combining them together into a final image.

FIGURE 6.2.16 From top to bottom, three views of two datasets showing the quality variation of the ambient occlusion when using 4/4, 8/8, and 16/16 directions and steps. The blur kernel size has been adjusted accordingly to compensate for the lack of samples.

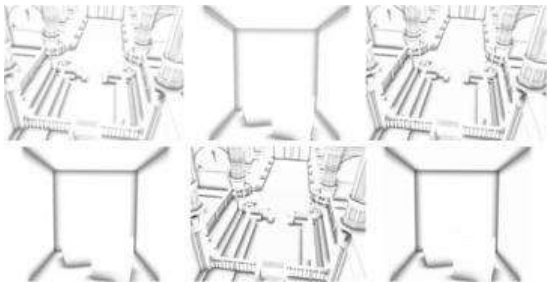


FIGURE 6.2.17 From top to bottom, diffuse shading on the dragon model, ambient occlusion calculated using our approach, and the final composited image.

ACKNOWLEDGMENTS

We thank Rouslan Dimitrov for the fruitful discussions in the early stages of our ambient occlusion research. We also thank Jason Mitchell, Naty Hoffman, and Nelson Max for the helpful discussions during the I3D '08 conference.

REFERENCES

[Baboud06] Baboud, L. and Decoret, X. “Rendering geometry with relief textures.” In GI ’06: Proceedings of Graphics Interface 2006, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 195–201, 2006.

[Bavoil08] Bavoil, L., Sainz, M., Dimitrov, R. “Image-space horizon-based ambient occlusion.” ACM SIGGRAPH 2008 talks, 2008.

[Cantlay07] Cantlay, I. “High-Speed, Off-Screen Particles.” GPU Gems 3, 2007.

[Christensen03] Christensen, P. H. “Global illumination and all that.” In ACM SIGGRAPH Course 9 (RenderMan, Theory and Practice), 2003.

[Dachsbacher07] Dachsbacher, C. and Tatarchuk, N. “Prism parallax occlusion mapping with accurate silhouette generation.” Symposium on Interactive 3D Graphics and Games, Poster, 2007.

[Davis07] Davis, S. T. and Wyman, C. “Interactive refractions with total internal reflection.” In GI ’07: Proceedings of Graphics Interface 2007.

[Dimitrov08] Dimitrov, R., Bavoil, L., and Sainz, M. “Horizon split ambient occlusion.” Symposium on Interactive 3D Graphics and Games, Poster, 2008.

[Eisemann04] Eisemann, E. and Durand, F. “Flash photography enhancement via intrinsic relighting.” ACM Transactions on Graphics (Proceedings of SIGGRAPH Conference), Volume 23, 2004.

[Gritz06] Gritz, L. Gelato 2.1 technical reference. Tech. rep., NVIDIA, 2006.

[Landis02] Landis, H., “Production Ready Global Illumination.” ACM SIGGRAPH Course 16: Renderman in Production, 2002.

[Langer99] Langer, M., and Bulthoff, H., “Perception of Shape From Shading on a Cloudy Day.” Technical Report, 1999.

[Lischinski98] Lischinski, D. and Rapoport, A. “Image-based rendering for non-diffuse synthetic scenes.” Proc. Ninth Eurographics Workshop on Rendering, in Rendering Techniques ‘98, pp. 301–314, 1998.

[Max86] Max, N. L. “Horizon mapping: shadows for bump-mapped surfaces.” In Proceedings of Computer Graphics Tokyo ’86 on Advanced Computer Graphics, 1986.

[Mittring07] Mittring, M., “Finding next gen: CryEngine 2.” In ACM SIGGRAPH 2007 courses, 2007, 97–121.

[Moller08] Akenine-Möller, T., Haines, E., and Hoffman, N. *Real-Time Rendering* (third edition). 2008.

[Musgrave88] Musgrave, F. K. “Grid tracing: fast ray tracing for height fields.” Technical Report YALEU/DCS/RR-639, Yale University, Dept. of Computer Science Research, 1988.

[Perlin89] Perlin, K. and Hoffert, E. “Hypertexture.” *Computer Graphics* (Proceedings of ACM SIGGRAPH Conference), Vol. 23, No. 3, 1989.

[Pharr04] Pharr, M. and Green, S., “Ambient Occlusion.” GPU Gems, 2004.

[Hegeman06] Hegeman, K., and Premoze, S., Ashikhmin, M., Drettakis, G., “Approximate ambient occlusion for trees.” Symposium on Interactive 3D Graphics and Games, 2006.

[Policarpo05] Policarpo, F., Oliveira, M. M., and Comba, J. “Real-time relief mapping on arbitrary polygonal surfaces.” Symposium on Interactive 3D Graphics and Games, 2005, 155–162.

[Segovia06] Segovia B., Iehl J. C., Mitanchey, R., and Péroche, B. “Non-interleaved deferred shading of interleaved sample patterns.” Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2006.

[Shanmugam07] Shanmugam, P., and Arikan, O. “Hardware accelerated ambient occlusion techniques on GPUs.” Symposium on Interactive 3D Graphics and Games, 2007, 73–80.

[Fox08] Fox, M. and Compton, S. “Ambient Occlusive Crease Shading.” Game Developer

Magazine, March 2008.

[Rogers85] Rogers, D. “Procedural elements for computer graphics”, 1985.

[Sloan00] Sloan, P.-P. J. and Cohen, M. F. “Interactive horizon mapping.” In Proceedings of the Eurographics Workshop on Rendering Techniques 2000.

[Tatarchuk06] Tatarchuk, N. “Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering.” In ACM SIGGRAPH 2006 Courses, 2006, 81–112.

[Xie07] Xie, F., Tabellion, E., Pearce, A. “Soft Shadows by Ray Tracing Multilayer Transparent Shadow Maps.” Eurographics Symposium on Rendering, 2007.

[Yagel92] Yagel, R., Cohen, D., and Kaufman, A. “Discrete ray tracing.” IEEE Computer Graphics & Applications, 1992.

[Zhukov98] Zhukov, S., Inoes, A., and Kronin, G. “An ambient light illumination model.” In Rendering Techniques ’98, G. Drettakis and N. Max, Eds., Eurographics, 1998, 45–56.