Vol. [VOL], No. [ISS]: 1–13

Fast Precomputed Ambient Occlusion for Proximity Shadows

Mattias Malmer and Fredrik Malmer
Syndicate

Ulf Assarsson
Chalmers University of Technology
Nicolas Holzschuch
ARTIS-GRAVIR/IMAG INRIA

Abstract. Ambient occlusion is used widely for improving the realism of real-time lighting simulations. We present a new method for precomputed ambient occlusion, where we store and retrieve unprocessed ambient occlusion values in a 3D grid. Our method is very easy to implement, has a reasonable memory cost, and the rendering time is independent from the complexity of the occluder or the receiving scene. This makes the algorithm highly suitable for games and other real-time applications.

1. Introduction

An “ambient term” is commonly used in illumination simulations to account for the light that remains after secondary reflections. This ambient term illuminates areas of the scene that would not otherwise receive any light. In first implementations, ambient light was an uniform light, illuminating all points on all objects, regardless of their shape or position, flattening their features, giving them an unnatural look.

To counter this effect, *ambient occlusion* was introduced by [Zhukov et al. 98]. By computing the *accessibility* to ambient lighting, and using it to modulate the effects, they achieve a much better look. Ambient occlusion is widely used in special ef-

1086-7651/06 $0.50 per page

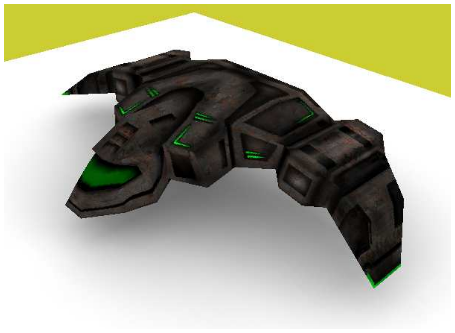


Figure 1. Example of contact shadows. This scene runs at 800 fps.

fects for motion pictures [Landis 02] and for illumination simulations in commercial software [Christensen 02, Christensen 03].

Ambient occlusion also results in objects having *contact shadows*: for two close objects, ambient occlusion alone creates a shadow of one object onto the other (see Figure 1).

For offline rendering, ambient occlusion is usually precomputed at each vertex of the model, and stored either as vertex information or into a texture. For real-time rendering, recent work [Zhou et al. 05, Kontkanen and Laine 05] suggest storing ambient occlusion as a field around moving objects, and projecting it onto the scene as the object moves. These methods provide important visual cues for the spatial position of the moving objects, in real-time, at the expense of extra storage. They pre-process ambient occlusion, expressing it as a function of space whose parameters are stored in a 2D texture wrapped around the object. In contrast, our method stores these *un-processed*, in a 3D grid attached to the object. The benefits are numerous:

- faster run-time computations, and very low impact on the GPU, with a computational cost being as low as 5 fragment shader instructions per pixel,
- very easy to implement, just by rendering one cube per shadow casting object,
- shorter pre-computation time,
- inter-object occlusion has high quality even for receiving points inside the occluding object's convex hull,
- handles both self-occlusion and inter-object occlusion in the same rendering pass.
- easy to combine with indirect lighting stored in environment maps.

The obvious drawback should be the memory cost, since our method's memory costs are in $O(n^3)$, instead of $O(n^2)$. But since ambient occlusion is a low frequency phenomenon, in only needs a low resolution sampling. In [Kontkanen and Laine 05], as in our own work, a texture size of $n = 32$ is sufficient. And since we are storing a single component per texel, instead of several function coefficients, the overall memory cost of our method is comparable to theirs. For a texture size of 32 pixels, [Kontkanen and Laine 05] report a memory cost of 100 Kb for each unique moving object. For the same resolution, the memory cost of our algorithm is of 32 Kb if we only store ambient occlusion, and of 128 Kb if we also store the average occluded direction.

2. Background

Ambient occlusion was first introduced by [Zhukov et al. 98]. In modern implementations [Landis 02, Christensen 02, Christensen 03, Pharr and Green 04, Bunnell 05, Kontkanen and Laine 05], it is defined as the percentage of ambient light blocked by geometry close to point $\boldsymbol{p}$:

$$ao(\boldsymbol{p}) = \frac{1}{\pi} \int_{\Omega} (1 - V(\omega)) \lfloor \boldsymbol{n} \cdot \omega \rfloor \, \mathrm{d}\omega \tag{1}$$

Occlusion values are weighted by the cosine of the angle of the occluded direction with the normal $\boldsymbol{n}$: occluders that are closer to the direction $\boldsymbol{n}$ contribute more, and occluders closer to the horizon contribute less, corresponding to the importance of each direction in terms of received lighting. Ambient occlusion is computed as a percentage, with values between 0 and 1, hence the $\frac{1}{\pi}$ normalization factor.

Most recent algorithms [Bunnell 05, Kontkanen and Laine 05] also store the average occluded direction, using it to modulate the lighting, depending on the normal at the receiving point and the environment.

[Greger et al. 98] also used a regular grid to store illumination values, but their grid was attached to the scene, not to the object. [Sloan et al. 02] attached radiance transfer values to a moving object, using it to recompute the effects of the moving object on the environment.

3. Algorithm

3.1. Description of the algorithm

Our algorithm inserts itself in a classical framework where other shading information, such as direct lighting, shadows, etc. are computed in separate rendering passes. One rendering pass will be used to compute ambient lighting, combined with ambient occlusion. We assume we have a solid object moving through a 3D scene, and we want to compute ambient occlusion caused by this object.

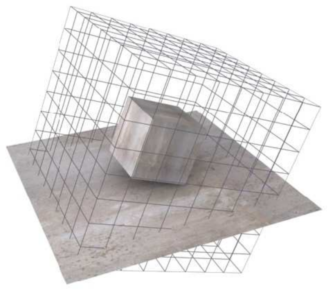


Figure 2. We construct a grid around the object. At the center of each grid element, we compute a spherical occlusion sample. At runtime, this information is used to apply shadows on receiving objects.

Our algorithm can either be used with classical shading, or with deferred shading. In the latter case, the world-space position and the normal of all rendered pixels is readily available. In the former, this information must be stored in a texture, using the information from previous rendering passes.

Precomputation: The percentage of occlusion from the object is precomputed at every point of a 3D grid surrounding the object (see Figure 2). This grid is stored as a 3D texture, linked to the object.

Runtime:

- render world space position and normals of all shadow receivers in the scene, including occluders.
- For each occluder:
  1. render the back faces of the occluder's grid (depth-testing is disabled).
  2. for every pixel accessed, execute a fragment program:
     (a) retrieve the world space position of the pixel.
     (b) convert this world space position to voxel position in the grid, passed as a 3D texture
     (c) retrieve ambient occlusion value in the grid, using linear interpolation.
  3. Ambient occlusion values a from each occluder are blended in the frame buffer using multiplicative blending with $1 - a$.

The entire computation is thus done in just one extra rendering pass. We used the back faces of the occluder's grid, because it is unlikely that they are clipped by the far clipping plane; using the front faces could result in artifacts if they are clipped by the front clipping plane.

3.2. Shading surfaces with ambient occlusion alone

The ambient occlusion values we have stored correspond to the occlusion caused by the occluder itself:

$$ao'(\boldsymbol{p}) = \frac{1}{4\pi}\int_{\Omega}(1 - V(\omega))\,\mathrm{d}\omega \tag{2}$$

that is, the percentage of the entire sphere of directions that is occluded. When we apply these occlusion values at a receiving surface, during rendering, the occlusion only happens over a half-space, since the receiver itself is occluding the other half-space. To account for this occlusion, we scale the occlusion value by a factor 2. This shading does not take into account the position of the occluder with respect to the normal of the receiver. It is an approximation, but we found it performs quite well in several cases (see Figure 1). It is also extremely cheap in both memory and computation time, as the value extracted from the 3D texture is used directly.

We use the following fragment program (using Cg notation):

```
float4 p_world = texRECT(PositionTex, p_screen)
float3 p_grid = mul(M_WorldToGrid, p_world)
out.color.w = 1 - tex3D(GridTexture, p_grid)
```

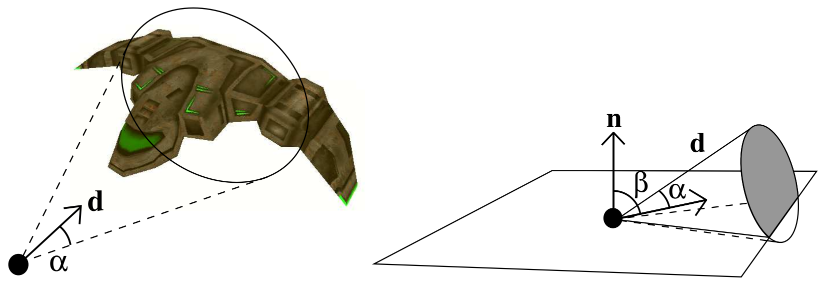
There are two important drawbacks with this simple approximation: first, the influence of the occluder is also visible where it should not, such as a character moving on the other side of a wall; second, handling self-occlusion requires a specific treatment, with a second pass and a separate grid of values.

3.3. Shading surfaces with ambient occlusion and average occluded direction

For more accurate ambient occlusion effects, we also store the average occluded direction. That is equivalent to storing the set of occluded directions as a cone (see Figure 3). The cone is defined by its axis ($\boldsymbol{d}$) and the percentage of occlusion a (linked to its aperture angle α). Axis and percentage of occlusion are precomputed for all moving objects and stored on the sample points of the grid, in an RGBA texture, with the cone axis $\boldsymbol{d}$ stored in the RGB-channels and occlusion value a stored in the A-channel.

3.3.1. Accounting for surface normal of receiver

In order to compute the percentage of ambient occlusion caused by the moving occluder, we clip the cone of occluded directions by the tangent surface to the receiver (see Figure 3(b)). The percentage of effectively occluded directions is a function of two parameters: the angle between the direction of the cone and the normal at the receiving surface (β), and the percentage of occlusion of the cone (a). We precompute this percentage and store it in a lookup table T_{clip}. The lookup table also stores



(a) The cone is defined by its direction ***d*** and its aperture α.

(b) The cone is clipped by the tangent plane to the receiver to give the ambient occlusion value.

Figure 3. Ambient occlusion is stored as a cone.

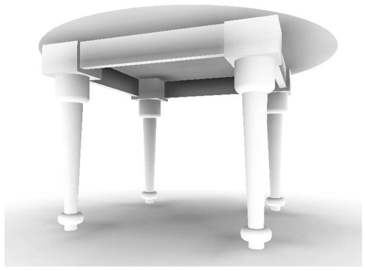


Figure 4. Ambient occlusion computed with our algorithm that accounts for the surface normal of the receiver and the direction of occlusion.

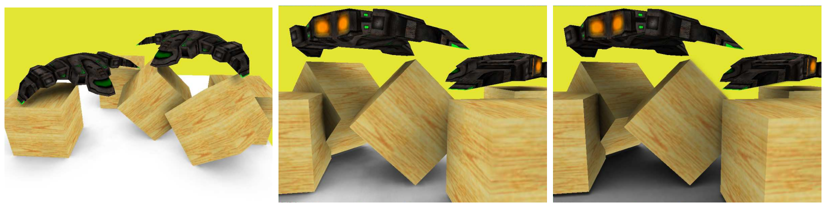
the effect of the diffuse BRDF (the cosine of the angle between the normal and the direction). For simplicity, we access the lookup table using $\cos\beta$.

We now use the following fragment program:

```
float4 p_world = texRECT(PositionTex, p_screen)
float3 p_grid  = mul(M_WorldToGrid, p_world)
float4 {d_grid, a} = tex3D(GridTexture, p_grid)
float3 d_world = mul(M_GridToWorld, d_grid)
float3 n = texRECT(NormalTex, p_screen)
float  cosβ = dot(d_world, n)
float  AO = texRECT(T_clip, float2(a, cosβ))
out.color.w = 1-AO
```

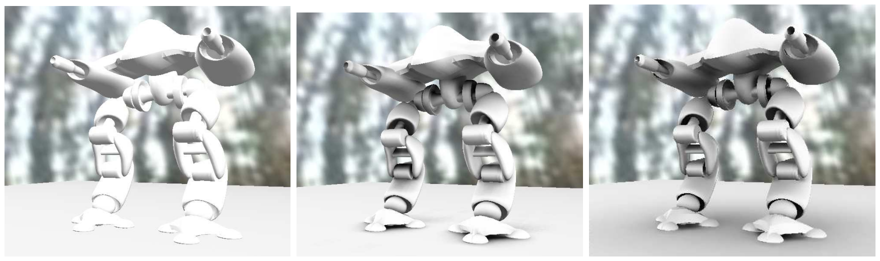
This code translates to 16 shader assembler instructions. Figure 4 and 5 were rendered using this method, with a grid resolution of 32^3.

Compared to storing only ambient occlusion values, using the average occluded direction has the advantage that results are more accurate and self-occlusion is naturally treated.



(a) (b) (c) Ground Truth

Figure 5. Ambient occlusion values, accounting for the normal of the occluder and the direction of occlusion (135 to 175 fps).



(a) Gouraud shading (b) Blending occlusion from multiple occluders (c) Ground truth

Figure 6. Checking the accuracy of our blending method: comparison of Ambient Occlusion values computed with ground truth.

3.3.2. Combining occlusion from several occluders

When we have several moving occluders in the scene, we compute occlusion values from each moving occluder, and merge these values together. The easiest method to do this is to use OpenGL blending operation: in a single rendering pass, we render the occlusion values for all the moving occluders. The occlusion value computed for the current occluder is blended to the color buffer, multiplicatively modulating it with $(1 - a)$.

[Kontkanen and Laine 05] show that modulating with $(1 - a_i)$, for all occluders i, is statistically the best guess. Our experiences also show that it gives very satisfying results for almost all scenes. This method has the added advantage of being very simple to implement: the combined occlusion value for one pixel is independent from the order in which the occluders are treated for this pixel, so we only need one rendering pass.

Each occluder is rendered sequentially, using our ambient occlusion fragment program, into an occlusion buffer. The cone axes are stored in the RGB channels and the occlusion value is stored in the alpha channel. Occlusion values are blended multiplicatively and cone axes are blended additively, weighted by their respective solid angle:

$$\begin{aligned}\alpha_R &= (1-\alpha_A)(1-\alpha_B) \\ \boldsymbol{d}_R &= \alpha_A \boldsymbol{d}_A + \alpha_B \boldsymbol{d}_B\end{aligned}$$

This is achieved using **glBlendFuncSeparate** in OpenGL. See Figures 5 and 6 for a comparison of blending values from several occluders with the ground truth values, computed with distributed ray-tracing: The two pictures exhibit the same important features, although our method is noticeably lighter (see also Section 4.3).

We have designed a more advanced method for blending the occlusions between two cones, taking into account the respective positions of the cones and their aperture (see the supplemental materials), but our experiments show that the technique described here generally gives similar results, runs faster and is easier to implement.

3.3.3. Illumination from an environment map

The occlusion cones can also be used to approximate the incoming lighting from an environment map, as suggested by [Pharr and Green 04]. For each pixel, we first compute the lighting due to the environment map, using the surface normal for Lambertian surfaces, or using the reflected cone for glossy objects. Then, we subtract from this lighting the illumination corresponding to the cone of occluded directions.

We only need to change the last step of blending the color buffer and occlusion buffer. Each shadow receiving pixel is rendered using the following code:

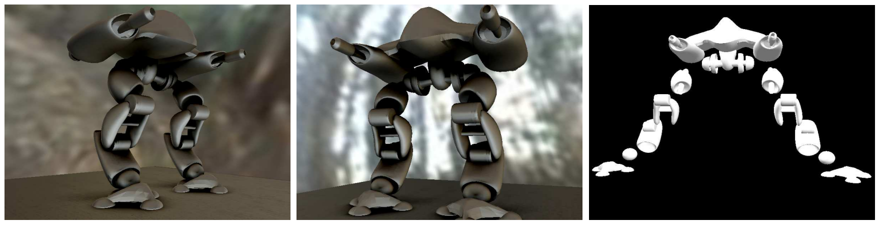
```
Pseudo code
1  Read cone $\boldsymbol{d}, \alpha$ from occlusion buffer
2  Read normal from normal buffer
3  Compute mipmap level from cone angle $\alpha$
4  A = EnvMap($\boldsymbol{d}, \alpha$). i.e., lookup occluded light within the cone
5  B = AmbientLighting(normal). i.e., lookup the incoming light due to the environment map.
6  return B-A.
```

In order to use large filter sizes, we used lat-long maps. It is also possible to use cube maps with a specific tool for mip-mapping across texture seams [Scheuermann and Isidoro 06].

3.4. Details of the algorithm

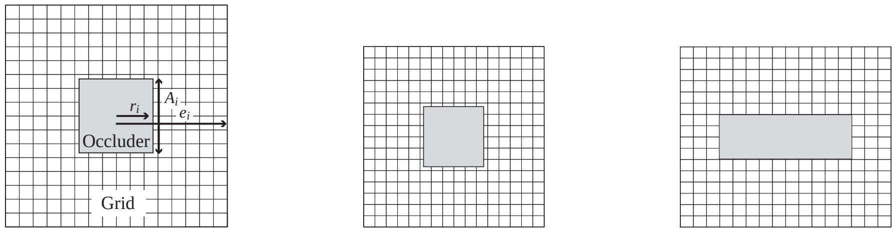
3.4.1. Spatial extent of the grid

An important parameter of our algorithm is the *spatial extent* of the grid. If the grid is too large, we run the risk of under-sampling the variations of ambient occlusion,



(a) (b) (c) Robot parts

Figure 7. Using ambient occlusion with environment lighting. These images are rendered in roughly 85 fps.



(a) Our notations (b) Cubic object (c) Elongated object. Notice the grid is *thinner* along the longer axis

Figure 8. Our notations for computing the optimal grid extent based on the bounding-box of the occluder (a), and optimal grid extents computed with $\epsilon = 0.1$ (b-c).

otherwise we have to increase the resolution, thus increasing the memory cost. If the grid is too small, we would miss some of the effects of ambient occlusion.

To compute the optimal spatial extent of the grid, we use the bounding box of the occluder. This bounding box has three natural axes, with dimension $2r_i$ on each axis, and a projected area of A_i perpendicular to axis i (see Figure 8(a)).

Along the i axis, the ambient occlusion of the bounding box is approximately:

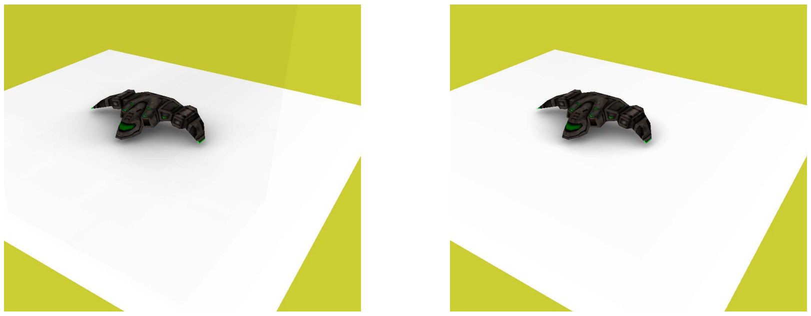
$$a_i \approx \frac{1}{4\pi} \frac{A_i}{(d - r_i)^2} \tag{3}$$

where d is the distance to the center of the bounding box.

If we decide to neglect occlusion values smaller than ϵ, we find that the spatial extent e_i of the grid along axis i should be:

$$e_i = r_i + \sqrt{\frac{A_i}{4\pi\epsilon}} \tag{4}$$

We take $\epsilon = 0.1$, giving an extent of $e_i \approx 3r_i$ for a cubic bounding box (see Figure 8(b)). For elongated objects, equation 4 gives an elongated shape to the grid,


(a) Using raw values, discontinuities can appear

(b) After re-scaling, ambient occlusion blends continuously

Figure 9. We need to re-scale occlusion values inside the grid to avoid visible artifacts.

following the shape of the object, but with the grid being thinner on the longer axes of the object (see Figure 8(c)).

We use a relatively large epsilon value (0.1), resulting in a small spatial extent. As a consequence, there can be visible discontinuities on the boundary of the grid (see Figure 9(a)). To remove these discontinuities, we re-scale the values inside the grid so that the largest value at the boundary is 0. If the largest value on the boundary of the grid is V_M, each cell of the grid is rescaled so that its new value V' is:

$$V' = \begin{cases} V & \text{if} \quad V > 0.3 \\ 0.3\frac{V-V_M}{0.3-V_M} & \text{if} \quad V \leq 0.3 \end{cases}$$

The effect of this scaling can be seen on Figure 9(b). The overall aspect of ambient occlusion is kept, while the contact shadow ends continuously on the border of the grid.

3.4.2. Voxels inside the occluder

Sampling points that are inside the occluder will have occlusion values of 1, expressing that they are completely hidden. As we interpolate values on the grid, a point located on the boundary of the occluder will often have non-correct values. To counter this problem, we modify the values inside the occluder (which are never used) so that the interpolated values on the surface are as correct as possible.

A simple but quite effective automatic way to do this is: for all grid cells where occlusion value is 1, replace this value by an average of the surrounding grid cells that have an occlusion value smaller than 1. This algorithm was used on all the figures in this paper.

4. Results

All timings and figures in this paper were computed on a Pentium 4, running at 2.8 GHz, with a NVidia GeForce 7800GTX, using a grid resolution of 32^3.

4.1. Timing results

The strongest point of our method is its performance: adding ambient occlusion to any scene increases the rendering time by $\approx$ 0.9 ms for each occluder. In our experiments, this value stayed the same regardless of the complexity of the scene or of the occluder. We can render scenes with 40 different occluders at nearly 30 fps.

The cost of the method depends on the number of pixels covered by the occluder's grid, so the cost of our algorithm decreases nicely for occluders that are far from the viewpoint, providing an automatic level-of-detail.

The value of 0.9 ms corresponds to the typical situation, visible in all the pictures in this paper: the occluder has a reasonable size, neither too small nor too large, compared to the size of the viewport.

4.2. Memory costs

Precomputed values for ambient occlusion are stored in a 3D texture, with a memory cost of $O(n^3)$ bytes. With a grid size of 32, the value we have used in all our tests, the memory cost for ambient occlusion values is 32 Kb per channel. Thus, storing just the ambient occlusion value gives a memory cost of 32 Kb. Adding the average occluded direction requires three extra channels, bringing the complete memory cost to 128 Kb.

4.3. Comparison with Ground Truth

Figure 5(b)-5(c) and 6(b)-6(c) show a side-by-side comparison between our algorithm and ground truth. Our algorithm has computed all the relevant features of ambient occlusion, including proximity shadows. The main difference is that our algorithm tends to underestimate ambient occlusion.

There are several reasons for this difference: we have limited the spatial influence of each occluder, by using a small grid, and the blending process (see Section 3.3.2) can underestimate the combined occlusion value of several occluders.

While it would be possible to improve the accuracy of our algorithm (using a more accurate blending method and a larger grid), we point out that ambient occlusion methods are approximative by nature. What is important is to show all the relevant features: proximity shadows and darkening of objects in contact, something our algorithm does.

Acknowledgments. ARTIS is an INRIA research project and a research team in the GRAVIR laboratory, a joint research unit of CNRS, INRIA, INPG and UJF.

This work was started while Ulf Assarsson was a post-doctoral student at the ARTIS research team, funded by INRIA.

The space ship model used in this paper was designed by Max Shelekhov.

References

[Bunnell 05] Michael Bunnell. "Dynamic Ambient Occlusion and Indirect Lighting." In *GPU Gems 2*, edited by Matt Pharr, pp. 223–233. Addison Wesley, 2005.

[Christensen 02] Per H. Christensen. "Note 35: Ambient occlusion, image-based illumination, and global illumination." In *PhotoRealistic RenderMan Application Notes*. Emeryville, CA, USA: Pixar, 2002.

[Christensen 03] Per H. Christensen. "Global Illumination and All That." In *Siggraph 2003 course 9: Renderman, Theory and Practice*, edited by Dana Batall, pp. 31 – 72. ACM Siggraph, 2003.

[Greger et al. 98] G. Greger, P. Shirley, P. M. Hubbard, and D. P. Greenberg. "The Irradiance Volume." *IEEE Computer Graphics and Applications* 18:2 (1998), 32–43.

[Kontkanen and Laine 05] Janne Kontkanen and Samuli Laine. "Ambient Occlusion Fields." In *Symposium on Interactive 3D Graphics and Games*, pp. 41–48, 2005.

[Landis 02] Hayden Landis. "Production Ready Global Illumination." In *Siggraph 2002 course 16: Renderman in Production*, edited by Larry Gritz, pp. 87 – 101. ACM Siggraph, 2002.

[Pharr and Green 04] Matt Pharr and Simon Green. "Ambient Occlusion." In *GPU Gems*, edited by Randima Fernando, pp. 279–292. Addison Wesley, 2004.

[Scheuermann and Isidoro 06] Thorsten Scheuermann and John Isidoro. "Cubemap Filtering with CubeMapGen." In *Game Developer Conference 2006*, 2006.

[Sloan et al. 02] Peter-Pike Sloan, Jan Kautz, and John Snyder. "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments." *ACM Transactions on Graphics (Proc. of Siggraph 2002)* 21:3 (2002), 527–536.

[Zhou et al. 05] Kun Zhou, Yaohua Hu, Steve Lin, Baining Guo, and Heung-Yeung Shum. "Precomputed Shadow Fields for Dynamic Scenes." *ACM Transactions on Graphics (proceedings of Siggraph 2005)* 24:3.

[Zhukov et al. 98] S. Zhukov, A. Iones, and G. Kronin. "An Ambient Light Illumination Model." In *Rendering Techniques '98 (Proceedings of the 9th EG Workshop on Rendering)*, pp. 45 – 56, 1998.

Web Information:

Two videos, recorded in real-time and demonstrating the effects of pre-computed ambient occlusion on animated scenes are available at:

`http://www.ce.chalmers.se/˜uffe/ani.mov`
`http://www.ce.chalmers.se/˜uffe/cubedance.mov`

A technique for better accuracy in blending the occlusion from two cones is described in a supplemental material.

Mattias Malmer, Syndicate, Grevgatan 53, 114 58 Stockholm, Sweden.
(www.syndicate.se)

Fredrik Malmer, Syndicate, Grevgatan 53, 114 58 Stockholm, Sweden
(www.syndicate.se)

Ulf Assarsson, Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden.
(uffe@ce.chalmers.se)

Nicolas Holzschuch, ARTIS/GRAVIR IMAG INRIA, INRIA Rhône-Alpes, 655 avenue de l’Europe, Innovallée, 38334 St-Ismier CEDEX, France.
(Nicolas.Holzschuch@inria.fr)

Received [DATE]; accepted [DATE].