INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fast Precomputed Ambient Occlusion for Proximity Shadows

Mattias Malmer — Fredrik Malmer — Ulf Assarsson — Nicolas Holzschuch

N° 5779

Décembre 2005

Thème COG

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5779--FR+ENG

Fast Precomputed Ambient Occlusion for Proximity Shadows

Mattias Malmer* , Fredrik Malmer* , Ulf Assarsson† ‡ , Nicolas Holzschuch‡

Thème COG — Systèmes cognitifs
Projets ARTIS

Rapport de recherche n° 5779 — Décembre 2005 —19 pages

Abstract: Ambient occlusion is used widely for improving the realism of real-time lighting simulations. We present a new, simple method for storing ambient occlusion values, that is very easy to implement and uses very little CPU and GPU resources. This method can be used to store and retrieve the percentage of occlusion, either alone or in combination with the average occluded direction. The former is cheaper in memory costs, while being slightly less accurate. The latter is slightly more expensive in memory, but gives more accurate results, especially when combining several occluders. The speed of our algorithm is independent of the complexity of either the occluder or the receiving scene. This makes the algorithm highly suitable for games and other real-time applications.

Key-words:

* Syndicate Ent., Grevgatan 53, SE-114 58 Stockholm, Sweden.
† Chalmers University of Technology, SE-412 96 Göteborg, Sweden.
‡ ARTIS/GRAVIR – IMAG INRIA Rhône-Alpes, France.

Fast Precomputed Ambient Occlusion for Proximity Shadows

Résumé : L'*Ambient Occlusion* est fréquemment utilisée pour améliorer le réalisme de simulations de l'éclairage en temps-réel. Nous présentons une nouvelle méthode pour stocker l'*Ambient Occlusion*. Cette méthode est simple et efficace, et n'utilise que très peu de ressources CPU et GPU. Elle peut être utilisée pour stocker et récupérer, soit le pourcentage d'occlusion seulement, soit le pourcentage d'occlusion combiné avec la direction moyenne de l'obstacle. La première méthode est moins couteuse en mémoire, mais la deuxième permet des calculs plus précis, en particulier lorsqu'on combine plusieurs obstacles ensemble. Notre algorithme est indépendant de la complexité, tant des obstacles que de la scène, ce qui en fait un algorithme très pratique pour les jeux et les applications temps-réel.

Mots-clés :

Figure 1: Example of our contact shadows. This scene runs at 200 fps.

1 Introduction

In illumination simulations, it has long been customary to add an "ambient term", to account for the light that remains after secondary diffuse reflections. The ambient term illuminates areas of the scene that would not otherwise receive any light. But ambient light is a uniform light: it illuminates all points on all objects, regardless of their shape or position. This gives objects illuminated mostly with ambient light an unnatural look, flattening their features.

To counter this effect, *ambient occlusion* was introduced by Zhukov *et al.* [ZIK98]. For every point in the scene, the accessibility to ambient lighting is computed and used to modulate the ambient light. Ambient occlusion is widely used in special effects for motion pictures [Lan02] and for illumination simulations in commercial softwares [Chr02, Chr03].

As a positive side effect, using ambient occlusion results in objects having *contact shadows*: when an object is close to another object, the ambient occlusion creates the shadow of the first object onto the second (see Fig. 1).

In early methods, ambient occlusion was precomputed at each vertex of the model, and stored either as vertex information or into a texture. In recent work [ZHL*05, KL05], targeting real-time rendering for video games, precomputed ambient occlusion is stored as a field around a moving object, and projected onto the scene as the object moves. Both these methods provide important visual cues on the relative positions of the moving object, in real-time, at the expense of storing extra information.

Both these methods *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, we suggest storing it *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 precomputation time,
- handles both self-occlusion and inter-object occlusion in the same rendering pass. In particular, the inter-object occlusion has high quality even for receiving points inside the occluding object's convex hull.
- easy to combine with indirect lighting stored in environment maps.

The most obvious drawback should be the memory cost, since our method's memory costs are in $O(n^3)$, compared to $O(n^2)$ for earlier work [ZHL*05, KL05]. But ambient occlusion is a low frequency phenomenon, stored in low resolution textures. In [KL05], as in our own work, a texture size of $n = 32$ is sufficient. And since we are storing a single component per texel, compared to several function coefficients, the overall memory cost of our algorithm is comparable to theirs. For a texture size of 32 pixels, Kontkanen and Laine [KL05] 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.

We present our method for storing and retrieving ambient occlusion values, with two applications: one where we only store and retrieve ambient occlusion values, and one where we also store the average unoccluded direction. The former is cheaper in memory costs, but the latter is more accurate, takes into account the normal at the receiving point and we show how to use it to compute the combined effects of several occluders and lighting from an environment map.

2 Background

Ambient light is used as an approximation of the untraced light in a scene. It is modeled as a light received uniformly at all points of a scene. As a result, objects that are illuminated only with ambient light (because direct or indirect lighting does not reach them) have a "flat" aspect, with all their 3D features erased.

To compensate for this, Zhukov *et al.* [ZIK98] suggested multiplying this value by *obscurance*, $w(\boldsymbol{p})$, defined as the accessibility to ambient light at each point $\boldsymbol{p}$:

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

In this equation, $V(\omega)$ is the visibility function, defined as 0 where there is some geometry occluding the view from point $\boldsymbol{p}$, and 1 where the direction of view ω is unoccluded. Objects that are closer to $\boldsymbol{p}$ should account more for the obscurance, while objects that are far away from $\boldsymbol{p}$ should account little, if anything. The function $\rho(d(\omega))$ weights the contribution of the occluder in direction ω by its distance ($d(\omega)$). Zhukov *et al.* [ZIK98] suggests using a smooth ρ function, limiting the contribution of occluders as they are further away from point $\boldsymbol{p}$. The integration is over the hemisphere centered around the normal at $\boldsymbol{p}$. From a physical point of view, this model can be thought of as an emitting gas, with constant emittance uniformly in all directions, present everywhere in the scene.

Zhukov *et al.* [ZIK98] defined obscurance as the percentage of ambient light that should reach each point $\boldsymbol{p}$. Modern implementations [Lan02, Chr02, Chr03, PG04, Bun05, KL05] reverse the meaning and define instead ambient occlusion as the percentage of ambient light that is *blocked* by geometry close to point $\boldsymbol{p}$. To simplify computations, they replace ρ with a step function, just computing the percentage of light that is blocked by occluders located within a given distance from $\boldsymbol{p}$:

$$ao(\boldsymbol{p}) = \frac{1}{\pi}\int_{\Omega} (1 - V(\omega))\lfloor \boldsymbol{n}\cdot\omega\rfloor \,\mathrm{d}\omega \tag{2}$$

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.

3 Basic algorithm

3.1 Overview of the algorithm

We assume we have a solid object moving through a 3D scene. We want to compute ambient occlusion caused by this object in the scene; our algorithm works as follows:

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

Runtime: For every pixel rendered, we get the ambient occlusion value in the 3D texture and use it. Retrieving ambient occlusion values is done by:

- rendering the back polygons of the bounding box of the grid,
- for each pixel rendered, get the world position of the underlying pixel,
- convert this position into voxel position in the grid,
- lookup ambient occlusion value (and, optionally, direction) in the 3D texture;

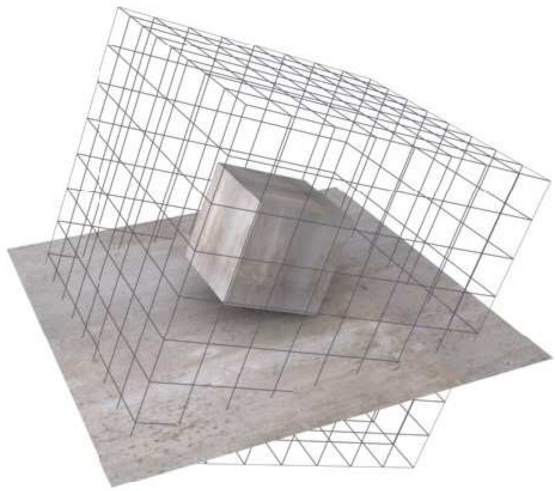


Figure 2: We construct a grid around the object. At the center of each grid element, we compute a spherical occlusion sample. At runtime, this information is used to apply shadows on receiving objects.

In short, for each moving occluder in the scene, the algorithm requires rendering three polygons and executing a very small fragment program at each pixel rendered. The cost on the GPU at runtime is really small, because we use un-processed values. Our method assumes that the position of the underlying pixel is available. This is naturally the case when using deferred shading, but it is also possible without deferred shading, *e.g.* by exporting the Z-buffer into a texture.

3.2 Detailed description of the algorithm

Our algorithm inserts itself in a classical framework where other shading information, such as direct lighting, shadows, etc. is computed in separate rendering passes. One rendering pass will be used to compute ambient lighting, combined with ambient occlusion.

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.

The overall algorithm is:

Step1. Render world space positions and normals: of all shadow receivers in the scene, including occluders. The normals will be used in section 5.1.

Step2. Ambient occlusion: for each occluder,

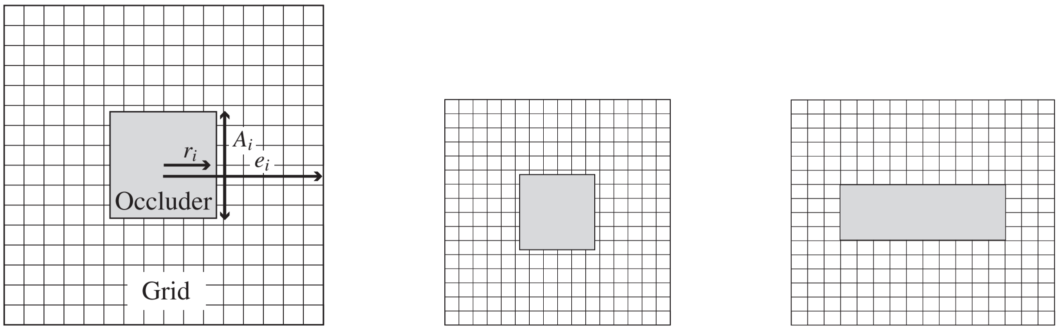
1. render the back faces of the occluder's grid (depth-testing is disabled).
2. for every pixel accessed, execute a fragment program:
   (a) retrieve the world space position of the pixel.
   (b) convert this world space position to voxel position in the grid, passed as a 3D texture
   (c) retrieve ambient 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.

The fragment program used in step 2 is shown here, in Cg:

FRAGMENT SHADER

```
float4 worldpos = texRECT(PositionTex, screenpos)
```



(a) Our notations

(b) Cubic object

(c) Elongated object. Notice the grid is thinner along the longer axis

Figure 3: The optimal grid extent depends on the bounding-box of the occluder. Here, grid extents are computed with $\varepsilon = 0.1$.

```
float3 gridpos.xyz = mul(WorldToGridMtx, worldpos)
out.color = tex3D(GridTexture, gridpos.xyz)
```

3.3 Details of the algorithm

3.3.1 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, a typical value we use in our implementation, the memory cost for ambient occlusion values is 32 Kb per channel. Thus, storing just the ambient occlusion value, as in section 4, gives a memory cost of 32 Kb. Adding the average occluded direction, as in section 5, requires three extra channels, bringing the complete memory cost to 128 Kb.

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

To compute the 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 Fig. 3(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:

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

We take $\varepsilon = 0.1$, giving an extent of $e_i \approx 3r_i$ for a cubic bounding box (see Fig. 3(b)). For elongated objects, equation 4 gives an elongated shape to the grid, following the shape of the object, but with the grid being thinner on the longer axes of the object (see Fig. 3(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 Fig. 4(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:

$$V' = \frac{V - V_M}{1 - V_M}$$

(a) Using raw values, discontinuities can appear

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

Figure 4: We need to re-scale occlusion values inside the grid to avoid visible artifacts.

The effect of this scaling can be seen on Fig. 4(b). The overall aspect of ambient occlusion is kept, while the contact shadow ends continuously on the border of the grid. This also resembles Zhukov's *et al.* [ZIK98] weighting with distance, $\rho(d(\omega))$, in equation 1.

3.3.3 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 less than 1. This algorithm was used on all the figures in this paper, except for the dwarf in Fig. 12 where we obtained better results with a little tweaking by hand.

4 Shading surfaces with ambient occlusion alone

4.1 Standard surfaces

In the "classical" ambient occlusion method, occlusion values are computed at a point $\boldsymbol{p}$ on a surface, with respect to the normal $\boldsymbol{n}$ at that point. The ambient occlusion is defined as the occlusion in the upper hemisphere, centered around the normal. Here, we are computing occlusion values at sample points in the grid, in a void, without any surface or normal. As a consequence, we have to change the definition of occlusion:

$$ao'(\boldsymbol{p}) = \frac{1}{4\pi}\int_{\Omega}(1 - V(\omega))\,\mathrm{d}\omega \tag{5}$$

This is the percentage of the full sphere that is occluded by our object. 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. This is an approximation, but we found it performs quite well in several cases (see Fig. 1). It is also extremely cheap in both memory and computation time, as the value extracted from the 3D texture is used directly.

In section 5, we explain a modification of our algorithm to account for the average occluded direction.

4.2 Self-occlusion

Our algorithm can also handle self-occlusion: ambient occlusion caused by one object onto itself. Self-occlusion will happen for non-convex objects, such as a table (Figure 7) or a character (Figure 12(a)). In that case, the occlusion value computed includes the underlying surface. Before using this value, we have to remove the effect of the underlying surface. Usually, the underlying surface occludes a full hemisphere. As a consequence, we have to use $ao'(\boldsymbol{p}) - 0.5$.

If we want to account for self-occlusion, depending on the surface being shaded, we have to use either $2ao'(\boldsymbol{p})$ or $ao'(\boldsymbol{p}) - 0.5$. For simplicity, we store:

$$ao_{grid}(\boldsymbol{p}) = 2ao'(\boldsymbol{p})$$

and when there is self-occlusion, we use:

$$ao_{self}(\boldsymbol{p}) = \frac{ao_{grid}(\boldsymbol{p}) - 1}{2}$$

We coded this using the stencil buffer to test for self-occlusion, and used a separate 3D texture to store the occlusion values ao_{self}.

5 Shading surfaces with ambient occlusion and average occluded direction

For better ambient occlusion effects, we also store the average occluded direction, along with the percentage of occlusion. That is equivalent to storing the set of occluded directions as a cone (see Fig. 5). This cone will be used for better accuracy in computing ambient occlusion, combining ambient occlusion from several occluders and computing environment lighting.

The cone is defined by its axis ($\boldsymbol{d}$) and the percentage of occlusion α (proportional to the solid angle covered by the cone). 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 α stored in the A-channel.

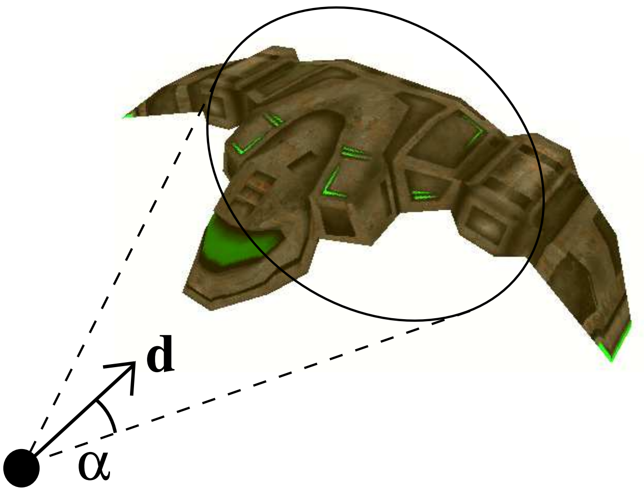


Figure 5: Ambient occlusion is stored as a cone, defined by its direction $\boldsymbol{d}$ and its aperture α

5.1 Accounting for surface normal of receiver

Compared to the algorithm in section 4, we now have access to more information regarding the relative positions of the occluder and receiver (see Fig. 6). 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 Fig. 6).

In step 2 of the algorithm (see Section 3.2), we have access to the world space position and normal of the underlying pixel. The world space position is used to access the cone direction and aperture in the grid. The normal is used to clip the cone of occluded directions: the surface at the pixel is approximated by its tangent plane, and we compute the intersection between this plane and the cone of occluded directions. The percentage of effectively occluded directions is a function of two parameters: the angle between the direction of the cone and the normal of the plane (β), and the aperture of the cone (α) (see Fig. 6). It is possible to compute this occlusion percentage, but for performance reasons we pre-compute it and store it in a lookup table, T_{clip}. The lookup table also stores the effect of the diffuse BRDF (the cosine of the angle between the normal and the direction).

The fragment shader code, in Cg notation:

FRAGMENT SHADER

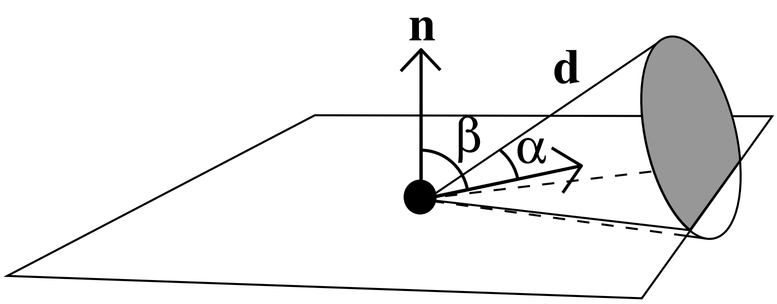


Figure 6: The cone of occluded directions is clipped by the tangent plane to the receiver to give the ambient occlusion value.

float4 $\mathbf{p}_{world}$ = texRECT(PositionTexture, $\mathbf{p}_{screen}$)
float3 $\mathbf{p}_{grid}$ = mul(WorldToGridMtx, $\mathbf{p}_{world}$)
float4 $\{\mathbf{d}_{grid}, \alpha\}$ = tex3D(GridTexture, $\mathbf{p}_{grid}$)
float3 $\mathbf{d}_{world}$ = mul(GridToWorldMtx, $\mathbf{d}_{grid}$)
float3 $\boldsymbol{n}$ = texRECT(NormalTexture, $\mathbf{p}_{screen}$)
float $\cos\beta$ = dot($\mathbf{d}_{world}$,$\boldsymbol{n}$)
float amb_occl = texRECT(T_{clip}, float2(α, $\cos\beta$))
out.color.w = 1-amb_occl

This code translates to 16 shader assembler instructions. Figure 7 and 8 were rendered using this method, with a grid resolution of 32^3. Note that, compared to the method described in section 4, this method handles self-occlusion at no extra charge; in the case of self-occlusion, the normal of the receiver is generally opposed to the axis of the cone. As a consequence, T_{clip} removes the self occlusion behind the tangent plane and only returns the self occlusion above the tangent plane, *e.g.* when $\alpha > 180°$.

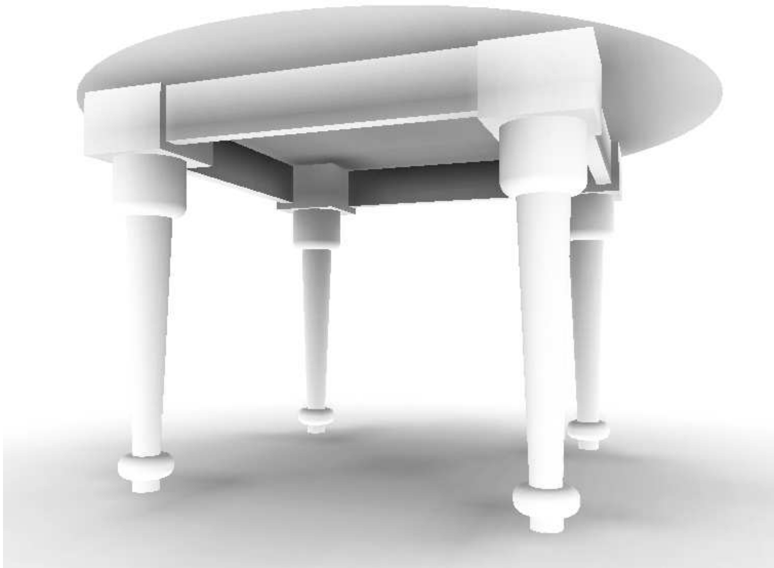
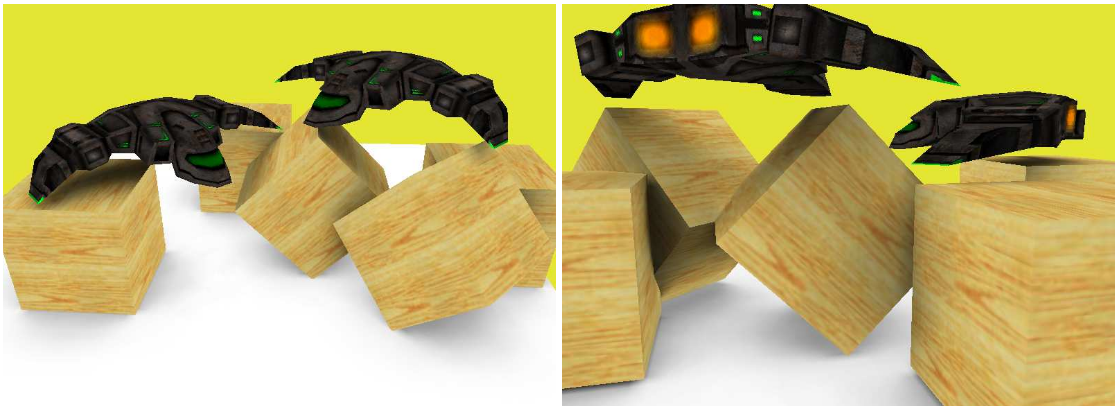


Figure 7: Ambient occlusion computed with our algorithm that accounts for the surface normal of the receiver and the direction of occlusion.

5.2 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



(a) (b)

Figure 8: Ambient occlusion values, accounting for the normal of the occluder and the direction of occlusion (80 to 130 fps)

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 [KL05] show that modulating with $(1-a_i)$, for all occluders i, is statistically the best guess. Our experience also show that, for many scenes, it gives very satisfying results.

This method also has the 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 it only requires one rendering pass to compute ambient occlusion from all moving occluders.

As we also need the axis of the occlusion cone, along with its aperture, we have to adapt the blending method. The key idea is that we treat occluders sequentially, keeping a cone corresponding to the union of all previous occlusion cones, at each step computing the union of this cone with the occlusion cone from the current occluder, and storing the result as input for the next occluder.

The trouble is that accurately combining one cone of occluded directions with another is a complex operation, which is not feasible using the simple blending operations currently available. It would benefit from either programmable blending or the ability to read and write into the same buffer.

We tested two possible workarounds. The first is the faster and the second is the more accurate:

- blending separately the apertures and the axes of the cone (section 5.2.1),
- keeping two separate buffers, one to be used for reading and the other for writing. After each occluder has been treated, the content of the second buffer is copied to the first. This slows down the computation, but ensures greater accuracy (section 5.2.2).

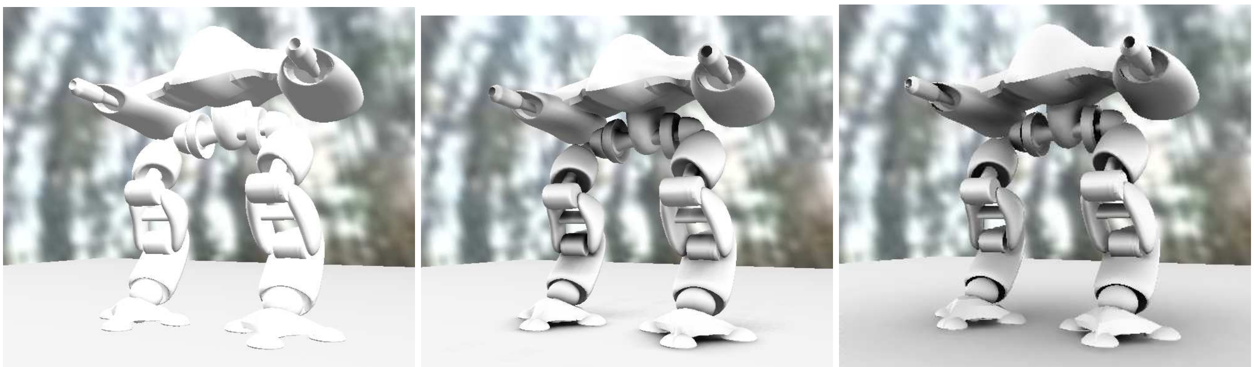
5.2.1 Simple occlusion blending

This method produces the same ambient occlusion values as the simple occlusion blending, but also outputs cone axes to be used for illumination from environment maps.

Each occluder is rendered sequentially, with the fragment shader in section 5.1, 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 still blended multiplicatively, while 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()**. Like the simple blending method, for this method the occlusion result is independent from the order in which the occluders are treated for this pixel, so it requires only one rendering pass for all the occluders. The resulting cone axes are normalized before usage with environment lighting.



(a) Gouraud shading

(b) Ambient Occlusion with simple blending

(c) Ground truth

Figure 9: Checking the accuracy of our blending method: comparison of Ambient Occlusion values computed with ground truth. See Figure 13(c) for a view of the robot parts.

Figure 9 shows a comparison of the ambient occlusion values computed by the simple blending method with the ground truth ambient occlusion values, computed by distributed ray-tracing. The ambient occlusion pictures look very similar, showing that even the simplest blending method is quite accurate for blending contributions from different parts.

5.2.2 Two Buffers Blending

The accuracy of blending occlusion cones can be improved by using two separate buffers, one *occlusion buffer* to be used for reading and one *copy-back buffer* for writing.

All the moving occluders are treated sequentially. For each occluder, we perform the following tasks:

1. Computing the occlusion cone using the algorithm described in section 5.1,
2. reading the occlusion cone from the occlusion buffer, corresponding to the combined effect of all the previous occluders, and
3. combining the two occlusion cones together and write the result into the copy-back buffer.

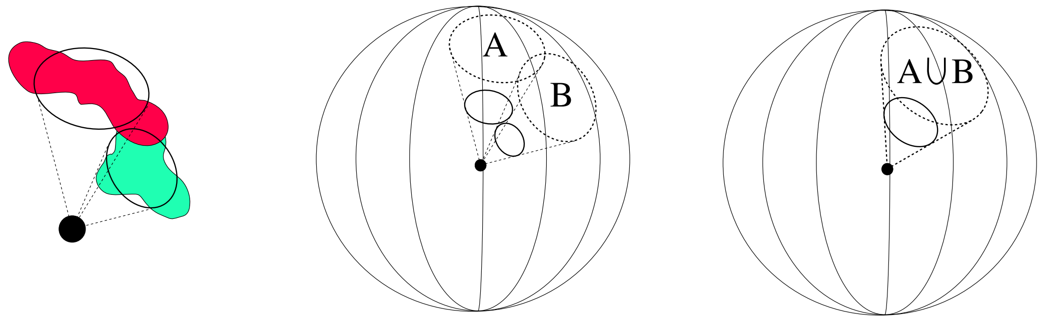
After dealing with the occluder, the content of the copy-back buffer is copied to the occlusion buffer, and we move on to the next occluder. Waiting until the treatment of the occluder is finished and copying occlusion values between the buffers slows down the computation.

After each rendering pass, each pixel contains a cone corresponding to the computed ambient occlusion at this pixel, which can be either a null cone, the cone from a single moving occluder, or a cone coming from the combination of several moving occluders.

Finally, when the contribution from all casters have been rendered, the alpha-channel of the occlusion buffer is blended onto the color buffer containing the colors of the receivers, using $1-a$.

Computing the union of two cones: Figure 10 illustrates combining together two ambient occlusion cones, each of them defined by an axis and an aperture. In the general case, the two cones are not disjoint, so we cannot satisfy ourselves with summing their occluded areas and averaging their central directions. Instead, we use a lookup table $T_{merge}(\alpha_1, \alpha_2, \beta)$ that returns the union of the two ambient-occlusion cones, taking their possible intersection into account. Here, the α_i represent the apertures of each cone, and β corresponds to the angle between the directions of the two cones. The two cones are then replaced by a new cone, whose aperture is given by T_{merge}, and whose axis is the weighted average of the two axes, weighted by their respective solid angle.

Clipping a cone against the tangent plane: When we clip an ambient occlusion cone against the tangent plane at the current receiving pixel, we compute the aperture of the clipped cone using the T_{clip} lookup function, described in



(a) (b) (c)

Figure 10: The occlusion from two separate shadow casters (a) are combined by replacing their two cones (b) by a resulting new cone, representing the occlusion of their union (c).

section 5.1. We then compute the new cone's central axis $\boldsymbol{d}_{A'}$ as a linear interpolation between the original cone central axis $\boldsymbol{d}_A$ and the normal to the tangent plane $\boldsymbol{n}$:

$$\boldsymbol{d}_{A'} = f\boldsymbol{d}_A + (1-f)\boldsymbol{n}$$

The blending coefficient, f, is also pre-computed as a function of (α, β) and is stored in the T_{clip} lookup texture.

Fragment shader: The fragment shader from section 5.1 needs the following extension:

FRAGMENT SHADER

7 **float2** $(\alpha_{A'}, f)$ = texRECT(T_{clip}, float2(α_A, β))
8 **float3** $\boldsymbol{d}_{A'}$ = normalize(lerp($\boldsymbol{d}_A$, n, f))
9 **float4** $(\boldsymbol{d}_B, \alpha_B)$ = texRECT(OcclusionBuffer, $\mathbf{p}_{screen}$)
10 out.color.xyz = $\boldsymbol{d}_{A'} * \alpha_A + \boldsymbol{d}_B$ //$\boldsymbol{d}_B$ premultiplied
11 **float** α_3 = dot($\boldsymbol{d}_{A'}$, normalize($\boldsymbol{d}_B$))
11 out.color.w = tex3D(T_{merge}, float3($\alpha_{A'}, \alpha_B, \alpha_3$))

Note on line 10, that $\boldsymbol{d}_B$ is stored in the occlusion buffer as premultiplied with α_B. out.color.xyz is the major direction of occlusion at the pixel, and out.color.w is the full sphere occlusion value.

(a) Adding occlusion

(b) Ground truth

(c) With our algorithm for combining occlusion from multiple objects

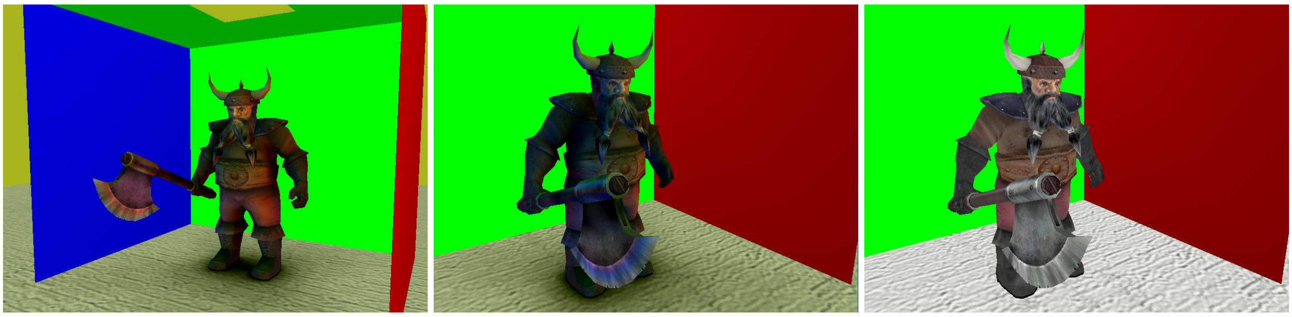
Figure 11: Two overlapping boxes and ambient occlusion result using different combining strategies.

Results: Figure 11 compares different methods for combining occlusion from two moving occluders. It shows two overlapping cubes, and their contact shadow. Figure 11(b) shows the ground truth, while Figure 11(a) shows the picture obtained by simply adding the occlusion from each shadow caster and Figure 11(c) shows the results obtained with our two-buffers blending method.

In Figure 11(a), the shadow is too dark under the place where the two cubes are overlapping, because the shadow from each cube are added together. Figure 11(c) avoids this pitfall, but the combined shadow is now slightly too light: representing the occlusion from each cube as a cone results in a loss of precision.

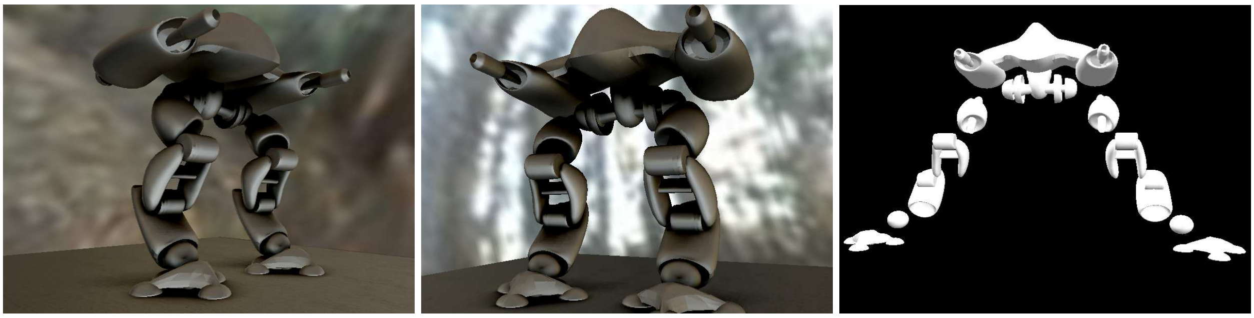
Animated rigid body objects (see Figures 13 and 9) typically have many intersecting parts or mutually occluding parts. In that case, accurately blending the occlusion from the moving parts is important, and the technique described in this section can be useful.

5.3 Accounting for illumination from an environment map



(a) (b) (c)

Figure 12: Using environment lighting: a dwarf model in a Cornell box. a) Notice the red color bleeding, and the dark shadow under his arm, where no color bleeding reaches from the red wall. b) The blue wall bleeds onto the dwarfs right side. c) Without ambient occlusion nor color bleeding.



(a) (b) (c)

Figure 13: Figures a) and b) demonstrates ambient occlusion together with environment lighting. These two images render in 30 fps. c) shows the 11 animated parts of the robot, here rendered with classic Gouraud shading. One ambient occlusion grid was created for each part.

The occlusion cones can also be used to approximate the incoming lighting from an environment map, as suggested by Pharr and Green [PG04]. See Figure 12 for an example of environment lighting.

Compared to their method, because the ambient occlusion is stored in 3D grids attached to each moving rigid object, we can handle not only self occlusion but also occlusion for animated objects made of several rigid bodies, such as the robot in Figure 13. This method can even be used to handle indirect illumination with color bleeding and local occlusion.

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 lighting 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:

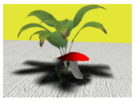
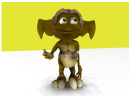
	#tris	GPU with occl queries	GPU with full cones	ray tracing
	391	0.45 s	2.75 s	5.58 s
	4.698	8.33 s	9 s	7.46 s
	16.271	50 s	50 s	6.25 s
	378k	230 s	230 s	92 s
	1.1M	-	-	226 s

Table 1: Preprocessing times for the 3D grid using three different method; 1) GPU acceleration and occlusion queries to only compute the ambient occlusion in each cell, 2) GPU acceleration computing occlusion cones, and 3) ray tracing with 256 rays per cell with Monte Carlo sampling

PSEUDO CODE

```
Read cone d, α from occlusion buffer
Read normal from normal buffer
Compute mipmap level from cone angle α
A = EnvMap(d, α). i.e., lookup occluded light within the cone
B = AmbientLighting(normal). i.e., lookup the incoming light due to the environment map.
return B-A.
```

To be able to use very large filter sizes, lat-long maps were used rather than cube maps, since mipmapping that spans multiple cube sides does not work well on current graphics hardware [PG04].

6 Results

6.1 Precomputation

We have implemented three different methods for precomputing the ambient occlusion grids: two methods using graphics hardware and one method based on ray tracing, shooting 256 rays per cell.

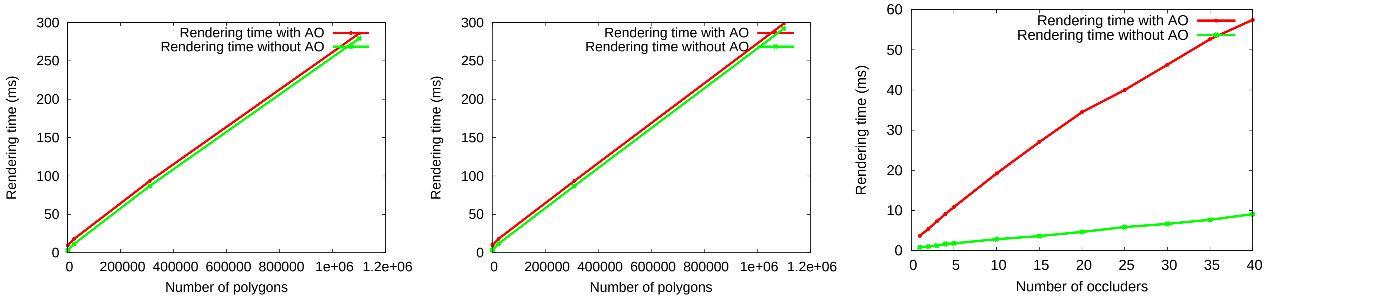
The GPU-based approaches renders the caster through the 6 cube sides of each cell, with a resolution of 16×16 per face. In the first method, we are only interested in occlusion values, so we use occlusion queries. In the second method, we are also looking for the direction of occlusion, so we read the Z-buffer to the CPU and average the occluded directions.

For all our scenes, a grid resolution of 32^3 was used. Table 1 shows the run-times. For comparison purposes, we have also included one very large model of 1M triangles.

The GPU-based versions are vertex bound and not fill-rate bound. Changing the resolution of the faces of the cubes from 8×8 up to 128×128 takes approximately the same time. Therefore, changing the projection method using parabolic or spherical projections should result in a 3 times speedup.

6.2 Timing results

Our tests were done on a Pentium4 2.8 GHz with a Geforce 6800GT. All images were rendered using a resolution of 800×600, and a grid resolution of 32^3. For all the timing results, we recorded the complete rendering time, that is the time for rendering the scene *and* computing ambient occlusion, not just computing ambient occlusion.



(a) Occluder Complexity (b) Scene Complexity (c) Number of occluders

Figure 14: Observed rendering times on our system, with and without Ambient Occlusion computations, depending on the parameters of the scene.

6.2.1 Occluder complexity

Figure 14(a) shows the evolution of rendering time (in ms) as a function of occluder complexity. We rendered the same scene, gradually increasing the complexity of the moving occluder and recording the rendering times, both with and without ambient occlusion.

As can be expected, the total rendering time increases with the occluder complexity. However, you can see that the added cost of using our pre-computed ambient occlusion is constant, roughly 6.5 ms, independently from the complexity of the occluder.

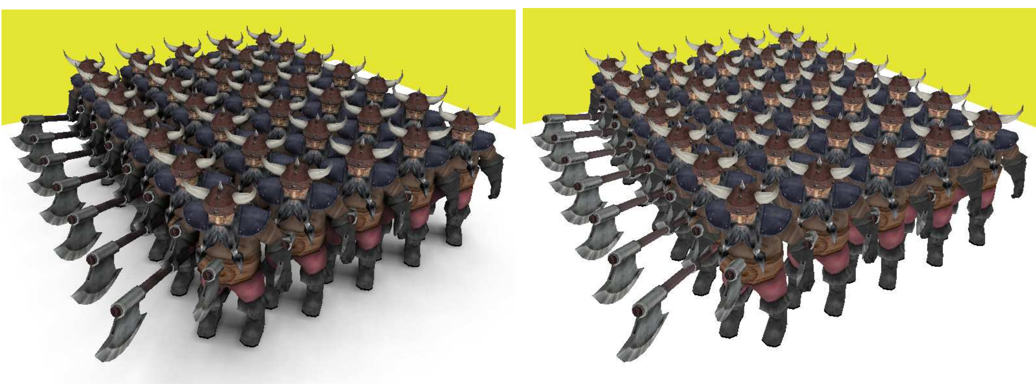
6.2.2 Scene complexity

Figure 14(b) shows the evolution of rendering time (in ms) as a function of scene complexity. We rendered the same occluder, inside a scene of increasing complexity, and recorded the rendering times, both with and without ambient occlusion.

Again, the total rendering time increases with scene complexity. However, the added cost of ambient occlusion is constant, roughly 6.5 ms, independently from the complexity of the occluder.

6.2.3 Number of occluders

Figure 14(c) shows the evolution of rendering time (in ms) as a function of the number of occluders. We rendered the same scene, gradually increasing the number of occluders and recorded the rendering times, both with and without ambient occlusion. The scene is shown in figure 15.



(a) With ambient occlusion (b) Without Ambient occlusion

Figure 15: The test scene for measuring the evolution of rendering time as a function of number of occluders.

Not surprisingly, the time used for ambient occlusion computations varies linearly with the number of occluders. The important point is the very low cost of our system: we are able to render up to 40 different moving occluders and still render at more than 17 fps.

6.2.4 Feature cost

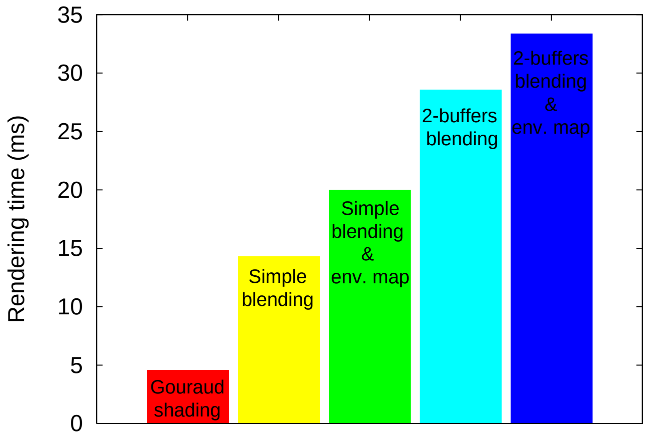


Figure 16: Observed rendering times on our system, depending on the features activated and on the blending method.

Figure 16 shows, for the robot scene, the evolution of rendering time depending on the activated features: rendering the scene with only standard Gouraud shading, with simple blending of ambient occlusion values, with simple blending and environment lighting, with two-buffers blending and finally with two-buffers blending and environment lighting.

Each feature improves the realism, but also has a cost in terms of rendering speed. All these costs are still manageable, however, and even with the best blending method and environment lighting, the 11 occluders in the robot are displayed at more than 30 fps.

On current architectures, the best compromise, in terms of rendering speed and aspect of the scene is to use the simple blending, combined with environment lighting.

6.3 Increasing scene realism

Figure 17 shows a classical test scene for real-time soft shadows [AAM03, HLHS03]. Due to the silhouette problem, most real-time soft shadow algorithms only see the top part of the box, and compute the soft shadow from this top part only. They are unable to compute the part of the soft shadow that is caused by occlusion from the bottom part of the box (see Figure 17(a)). Figure 17(b) shows a combination of our Ambient Occlusion algorithm with Soft Shadow Volumes, computed separately. As you can see, the resulting picture is much closer to the ground truth (Figure 17(c)), and the cost of adding ambient occlusion is almost negligible.

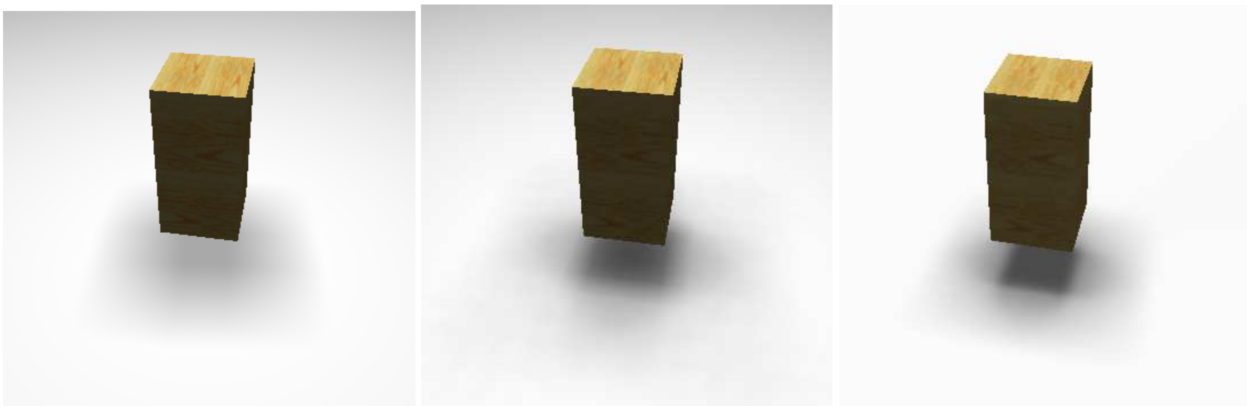
7 Acknowledgments

The space ship model used in this paper is a freeware model by Max Shelekhov.

References

[AAM03] ASSARSSON U., AKENINE-MÖLLER T.: A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003) 22*, 3 (July 2003), 511–520.

[Bun05] BUNNELL M.: Dynamic ambient occlusion and indirect lighting. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, 2005, pp. 223–233.



(a) Soft Shadow Volumes (75 fps) (b) Soft Shadow Volumes + Ambient Occlusion (68 fps) (c) Ground Truth (55 s)

Figure 17: Using Ambient Occlusion to increase the realism of soft shadows.

[Chr02] CHRISTENSEN P. H.: Note 35: Ambient occlusion, image-based illumination, and global illumination. In *PhotoRealistic RenderMan Application Notes*. Pixar, Emeryville, CA, USA, Apr. 2002.

[Chr03] CHRISTENSEN P. H.: Global illumination and all that. In *Siggraph 2003 course 9: Renderman, Theory and Practice*, Batall D., (Ed.). ACM Siggraph, 2003, pp. 31 – 72.

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F.: A survey of real-time soft shadows algorithms. *Computer Graphics Forum 22*, 4 (Dec. 2003), 753–774.

[KL05] KONTKANEN J., LAINE S.: Ambient occlusion fields. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games* (2005), ACM Press, pp. 41–48.

[Lan02] LANDIS H.: Production ready global illumination. In *Siggraph 2002 course 16: Renderman in Production*, Gritz L., (Ed.). ACM Siggraph, 2002, pp. 87 – 101.

[PG04] PHARR M., GREEN S.: Ambient occlusion. In *GPU Gems*, Fernando R., (Ed.). Addison Wesley, 2004, pp. 279–292.

[ZHL*05] ZHOU K., HU Y., LIN S., GUO B., SHUM H.-Y.: Precomputed shadow fields for dynamic scenes. *ACM Transactions on Graphics (proceedings of Siggraph 2005) 24*, 3 (2005).

[ZIK98] ZHUKOV S., IONES A., KRONIN G.: An ambient light illumination model. In *Rendering Techniques '98 (Proceedings of the 9th EG Workshop on Rendering)* (1998), pp. 45 – 56.

Contents

List of Figures

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399