ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΣΤΗΝ ΕΠΙΣΤΗΜΗ ΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία
Μεταπτυχιακού Διπλώματος Ειδίκευσης

«*Pre-calculated Volumetric Illumination for Real-time Graphics*»

Μενεξή Μαρία – Λήδα

ΕΥ0627

Επιβλέπων: Παπαϊωάννου Γεώργιος

ΑΘΗΝΑ, ΙΟΥΛΙΟΣ 2008

Master Thesis

Author: Maria Lida Menexi

2008, Athens Greece

Pre-calculated Volumetric Illumination for Real-time Graphics

Abstract. Real-time rendering can benefit from global illumination methods to make the three-dimensional scenes look more convincing and lifelike. On the other hand, the conventional global illumination algorithms make heave usage of intra- and inter-object visibility calculations, so they are very time consuming, and using them in real-time graphics is prohibitive. Modern illumination approximations, such as ambient occlusion variants, use pre-calculated data to reduce the problem to a local shading one. This thesis presents an alternative method for the pre-calculation and fast real-time exploitation of visibility information in order to produce a visually pleasant ambient occlusion and lighting approximation. Unlike other methods, ours does not use directional data. Therefore, it is a fast algorithm for real-time graphics that works well for both complex and simple geometry and uses a comparatively small amount of memory storage. The proposed algorithm can be used to illuminate arbitrary convex or concave surfaces and poses no limitation on surface connectivity.

Table of Contents

1. Introduction

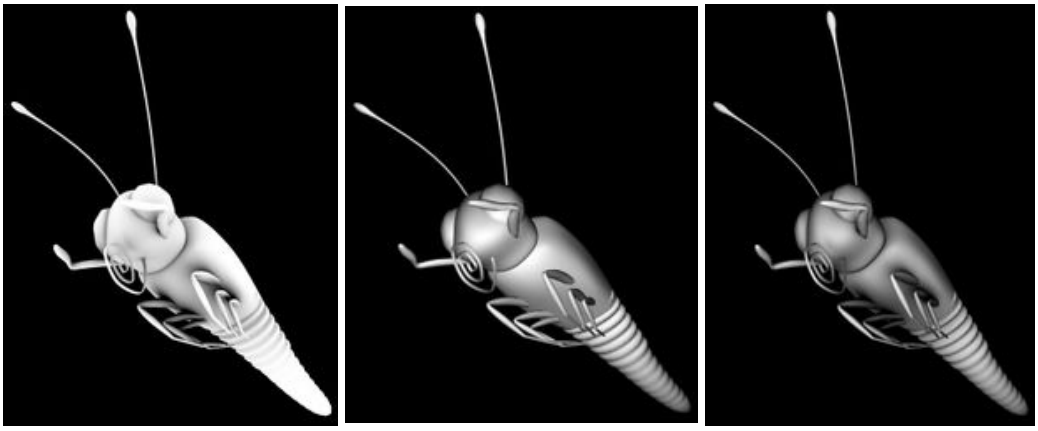
An “ambient term” is commonly used in illumination simulations to account for the lighting that is received by surfaces indirectly, after being diffusely relfected off other surfaces. This ambient term illuminates areas of the scene that would not otherwise receive any light. In first implementations, ambient light was a constant term, uniformly illuminating all points on all objects, regardless of their shape or position, flattening their features and giving them an unnatural look. To counter this effect, global illumination methods such as radiosity and photon tracing, were first introduced into non-real-time rendering and were successfully used in conjunction with lightmaps (pre-calculated static illumination) to improve the visual quality and credibility of stills and animations; the only drawback was the fact that all that was computationally very expensive. Ambient occlusion was introduced by [Zhuk98],as a fast approximation to the visual effect that the methods above resulted to, but half of the credit belongs to the movie industry and the production rendering community for refining and popularizing the technique.

1.1. Ambient Occlusion

Ambient occlusion is a shading method used in 3D computer graphics which helps add realism to local reflection models by taking into account attenuation of light due to occlusion. Unlike local methods like Phong shading, ambient occlusion is a global method, meaning the illumination at each point is a function of other geometry in the scene. However, it is a very crude approximation to full global illumination. The soft appearance achieved by ambient occlusion alone is similar to the way an object appears on an overcast day.

Ambient occlusion is most often calculated by casting rays in every direction from the surface. Rays which reach the background or “sky” increase the brightness of the surface, whereas a ray which hits any other object contributes no illumination. As a result, points surrounded by a large amount of geometry are rendered dark, whereas points with little geometry on the visible hemisphere appear light.

Ambient occlusion is related to accessibility shading, which determines appearance based on how easy it is for a surface to be touched by various elements (e.g., dirt, light, etc. – see *Fig. 1.1.1*). It has been popularized in production animation due to its relative simplicity and efficiency. The ambient occlusion shading model has the nice property of offering a better perception of the 3d shape of the displayed objects. This was shown in a paper where the authors report the results of perceptual experiments showing that depth discrimination under diffuse uniform sky lighting is superior to that predicted by a direct lighting model.



(a) ambient occlusion (b) diffuse only (c) combined ambient and diffuse

Fig. 1.1.1. The ambient occlusion effect

The occlusion $A_{\mathbf{p}}$ at a point **p** on a surface with normal $\vec{\mathbf{n}}$ can be computed by integrating the visibility function over the hemisphere Ω with respect to projected solid angle, as shown in *Fig. 1.1.2*:

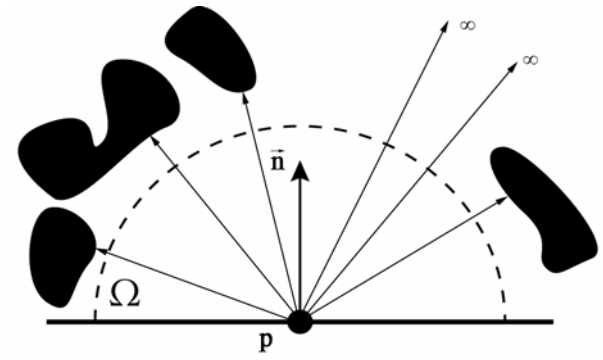


Fig. 1.1.2. The occlusion hemisphere

$$A_{\mathbf{p}} = \frac{1}{\pi}\int_{\Omega} \rho\left(d(\mathbf{p},\vec{\omega})\right)\left(\vec{\omega}\cdot\vec{\mathbf{n}}\right)d\omega \qquad (1.1.1)$$

where $d(\mathbf{p},\vec{\omega})$ refers to the distance of the first intersection when a ray is shot from **x** towards $\vec{\omega}$. ρ is a function that maps the distance suitably. $\rho\left(d(\mathbf{p},\vec{\omega})\right)$ represents the visibility function at ***p***, defined to be zero when no geometry is visible in the direction $\vec{\omega}$ and one otherwise. Iones et al. [Iones2003] suggest $1-e^{\tau d}$, where τ is a user defined constant. $\int_{\Omega}$ refers to integration over a hemisphere oriented according to the surface normal $\vec{\mathbf{n}}$.

A variety of techniques are used to approximate this integral in practice: perhaps the most straightforward way is to use the Monte Carlo method by casting rays from the point **p** and testing for intersection with other scene geometry (i.e., ray casting). But that is a quite slow procedure because for every point many rays have to be cast in order to get a decent result.

Another approach, that of "scattering" or "outside-in" techniques, is used in depth-map ambient occlusion algorithms.

1.2. Method Overview

We shall present an algorithm for pre-calculated visibility, that can be used to produce a visually pleasant ambient occlusion and lighting approximation. Unlike other methods that store directional visibility information for every location in space [Gait08], [Kont05], [Malm06], it does not use directional data, so it is a very fast method that uses a comparatively small amount of memory storage. Fourthermore, it does not suffer from view-dependency, nor has it the local effect depth-map-based ambient occlusion algorithms have [Shan06]. As will be discussed later, an interesting consequence of the proximity value storage used in our algorithm is that it is possible to incorporate in the same ambient illumination calculation the contribution of direct diffuse illumination, such as light emitters of various sizes, shapes and light emission distributions.

More specifically, first each object in the scene is voxelized, so a voxel grid is created around the object and the voxels intersected by its surface are marked as “internal” (maximum surface proximity – minimum distance). Then a proximity value is calculated for each voxel, with a flooding algorithm. In this algorithm, the currently calculated proximity value of every voxel is diffused to its neighbors via an iterative propagation scheme, so that each voxel eventually stores an approximate proximity value or, in other words, cumulative distance to the neighboring surfaces. This approach is faster than explicitly calculating the cumulative distance for each individual voxel by a Monte-Carlo method or any other used in Ambient Occlusion. The proximity values are stored in a 3D texture, and are passed on to a shader during real-time rendering. In the shader, for every fragment, the proximity values are sampled along the normal direction from the stored distance information, and based on that, an occlusion value is reconstructed. The proximity volumes are expressed in the local space of each object and during an animated sequence with multiple objects, the volumes are also transformed along with their associated objects. In order to account for the intra- and inter-object ambient shading, the cumulative contribution of the proximity volumes of all nearby objects is estimated.

As we have already mentioned, the main advantage of the proposed algorithm is that it does not use directional data (and consequently it doesn’t have to store such data), and most of the computations are done in the pre-processing stage; so this is a fast algorithm for real-time graphics, that uses a comparatively small amount of memory storage. The algorithm also has good scalability as the volume distance data that are generated in a pre-processing stage only depend on the new objects. Later on, we will show that the proposed algorithm works well for both complex and simple geometry and that it can be used to illuminate arbitrary convex or concave surfaces and also that it poses no limitation on surface connectivity.

In Section 2 we give an overview of the previous work, followed by a description of our method in greater detail in Section3. In Section 4 we discuss the results of the implementation of the proposed method, and finally, in Section 5 we reach some conclusions about this method.

2. RELATED WORK

2.1. Ambient Occlusion

2.1.1. Dynamic Ambient Occlusion and Indirect Lighting

A method for computing diffuse light transfer and how it can be used to compute global illumination for animated scenes, is presented in [Bunn05]. The first step of this algorithm is to convert the polygonal data to surface elements (*surfels*) to make it easy to calculate how much one part of a surface shadows or illuminates another (*Fig. 2.1.1)* so that a surface element is defined as an oriented disk with a position, normal, and area.

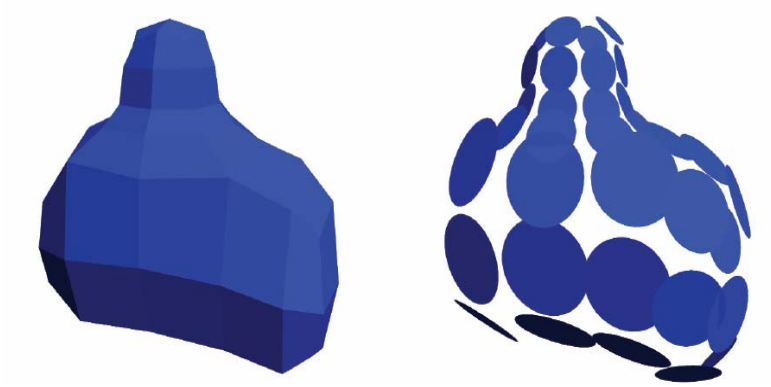


Fig. 2.1.1. Converting a Polygonal Mesh to Elements. Left: A portion of a polygonal mesh. Right: The mesh represented as disk-shaped elements.

In order to calculate ambient occlusion, accessibility values are needed first, which are calculated in 2 passes. The "bent normal" (the direction of least occlusion) is also required, which can be calculated during the second pass. In the first, the accessibility for each element is approximated by summing the solid angles subtended by every other element and subtracting the result from 1. After the first pass, some elements will generally be too dark because other elements that are in shadow are themselves casting shadows. So, a second pass is used to do the same calculation but this time to multiply each form factor by the emitter element's accessibility from the last pass. The effect is that elements that are in shadow will cast fewer shadows on other elements (*Fig 2.1.2*). More passes can be used to get a better approximation, but the same solution can be approximated by using a weighted average of the combined results of the first and second passes. The occlusion result is calculated by rendering a single quad (or two triangles) so that one pixel is rendered for each surface element. The shader calculates the amount of shadow received at each element and writes it as the alpha component of the color of the pixel. The results are rendered to a texture map so the second pass can be performed with another rendering pass. In this pass, the bent normal is calculated and written as the RGB value of the color with a new shadow value that is written in the alpha component.

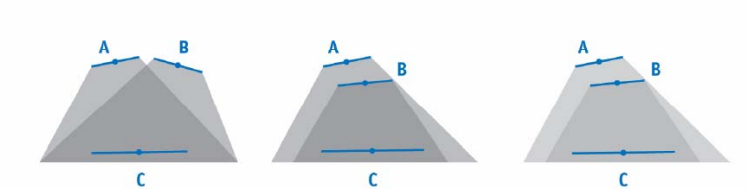


Fig. 2.1.2. Correcting for Occlusion by Overlapping Objects. Left: Elements A and B correctly shadow C after the first pass. Middle: In this arrangement, B casts too much shadow on C. Right: B's shadow is adjusted by the second pass to shadow C properly.

In order to improve performance, elements whose vertices are neighbours in the original mesh are grouped and represented with a single, larger element. The same thing can be done with the larger elements, creating fewer and consequently larger elements, forming a hierarchy (*Fig. 2.1.3*). Now instead of traversing every single element for each rendered pixel, the hierarchy of elements is traversed. If the receiver element is far enough away from the emitter—say, twice the radius of the emitter—it can be used for calculation, but if it is close to an emitter, its children need to be traversed (if it has any). This way, the performance of the algorithm can be improved from $O(n^2)$ to $O(n \log n)$ in practice. A parent element's data is calculated using its direct descendants in the hierarchy; its position and normal are calculated by averaging the positions and normals of its children and its area as the sum of its children's areas. For these calculations a shader can be used, making one pass for each level in the hierarchy, propagating the values from the leaf nodes up. The same technique can be used to average the results of an occlusion pass that are needed for a following pass or simply treat parent nodes the same as children and avoid the averaging step. It is worth noting that the area of most animated elements varies little, if at all, even for non-rigid objects; therefore, the area does not have to be recalculated for each frame.

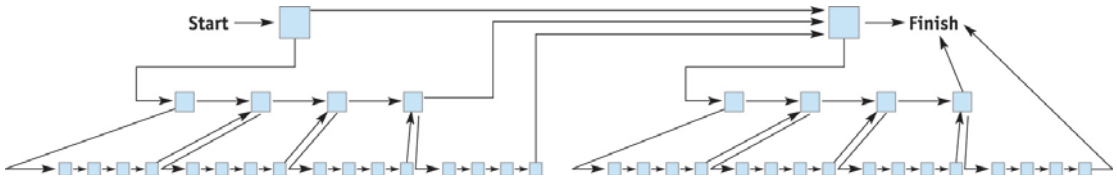


Fig. 2.1.3. Hierarchical Elements. Elements are traversed in a hierarchy; child elements are traversed only if necessary.

However, this method requires a huge pre-computation step with little support for dynamic objects, and requiring high-tessellation of scene geometry.

2.1.2. Ambient Occlusion Fields

A novel real-time technique for computing inter-object ambient occlusion, which partially solves the dynamic object problem of the previous method, is presented in [Kont05]. For each occluding object, a field in the surrounding space that encodes an approximation of the occlusion caused by the object is precomputed. This volumetric information is then used at run-time in a fragment program for quickly determining the shadow cast on the receiving objects. Results show that both the computational and storage requirements are low enough for the technique to be directly applicable to computer games running on the current graphics hardware. The occluder is approximated by a spherical cap when computing the ambient occlusion on the receiving surfaces. A spherical cap is defined by a direction and a solid angle (*Fig. 2.1.4*). To quickly

determine a spherical cap at an arbitrary location in the neighborhood, the subtended solid angle and the average direction of occlusion as fields around the occluder are precomputed. To keep the memory requirements low, these fields are stored as radial functions into a cube-map surrounding the occluder. The spherical cap approximation works well in practice, since the ambient occlusion integral is a smooth, cosine weighted average of the visibility function over a hemisphere.

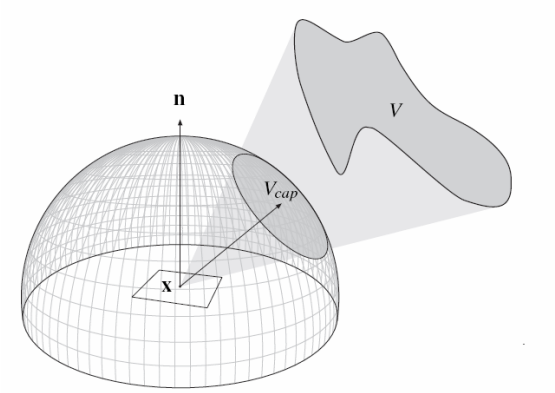


Fig. 2.1.5. A spherical cap approximation for an occluder. An approximate ambient occlusion is computed at each **x** with surface normal $\vec{\mathbf{n}}$ by approximating the visibility function V(**x**,**w**) by a visibility of a corresponding spherical cap V_{cap}(**x**,**w**). The size of the spherical cap is determined so that it subtends the same solid angle as the occluder. The direction of the cap is given by the average of the occluded directions.

To combine shadows from multiple casters efficiently, multiplicative blending 1-A for each occluder to the frame-buffer is proposed (where A is the approximated ambient occlusion value). If we call A_{ab} the ambient occlusion of 2 occluders, a and b, then an approximation for it is needed by utilizing the known ambient occlusion values A_a and A_b. There are three different cases about A_a and A_b that are illustrated in *Fig. 2.1.5*. When an occluder completely overlaps the other one, the combined ambient occlusion is given by picking up the larger of the values A_a and A_b. When the occluders do not overlap at all the value is given by the sum of the ambient occlusion terms of each object. It is easy to see that these two cases represent the extremes and the combined ambient occlusion A_{ab} always satisfies: $\max(A_a,A_b) \leq A_{ab} \leq A_a + A_b$. Multiplicative blending of $1-A$ for each object satisfies the above inequality.

In addition, it can be shown that a ray shot from a receiving surface with a cosine weighted probability distribution, hits occluding geometry with probability equal to ambient occlusion. Thus, if we understand A_x as probability of hitting object x, A_{ab} can be interpreted as probability of hitting either or both of the objects. Assuming that A_a and A_b are uncorrelated and combining the probabilities by elementary probability theory yields: $1-A_{ab} = (1-A_a)(1-A_b)$.

This suggests the multiplicative blending of the shadow casted by each occluder into the frame-buffer. Note that the result is probabilistic and thus an approximation of A_{ab}.

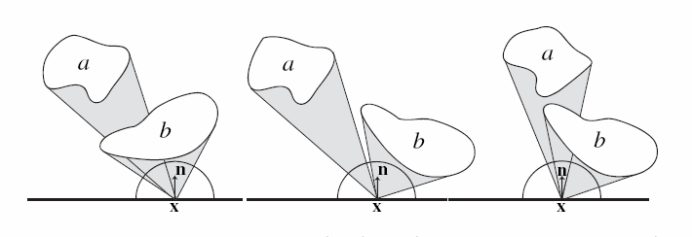


Fig. 2.1.5. A 2D-illustration of the three different ways two occluders may block point x on a receiving surface. Left: Object a is completely occluded by object b, and just picking up the bigger ambient occlusion, $\max(A_a,A_b)$, would yield the correct combined ambient occlusion A_{ab}. Middle: The occluders do not overlap each other, and the correct result would be given by adding the ambient occlusion terms of the objects: $A_{ab} = A_a+A_b$. Right: The interaction of the occluders is more complex, but the ambient occlusion is always between $\max(A_a,A_b)$ and $A_a + A_b$.

2.1.3. Fast Precomputed Ambient Occlusion for Proximity Shadows

The next method described is for real-time rendering and it suggests pre-processing ambient occlusion, expressing it as a function of space, storing ambient occlusion as a field around moving objects, just like the previous one. The main difference from the previous technique, is that this method, described in [Malm06], stores the un-processed data (ambient occlusion values) in a 3D grid attached to the object.

This 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. It is assumed that there is a solid object moving through a 3D scene, and we want to compute ambient occlusion caused by this object.

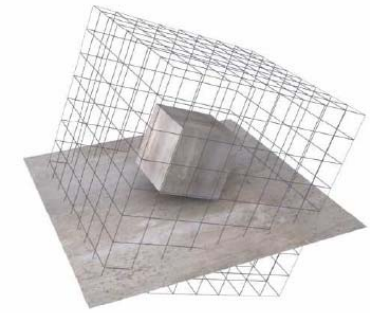


Fig. 2.1.6. A grid is constructed around the object. At the center of each grid element, a spherical occlusion sample is computed. At runtime, this information is used to apply shadows on receiving objects.

The outline of the algorithm is as follows:

Precomputation: The percentage of occlusion from the object is precomputed at every point of a 3D grid surrounding the object (see *Fig. 2.1.6*). This grid is stored as a 3D texture, linked to the object.

Runtime:

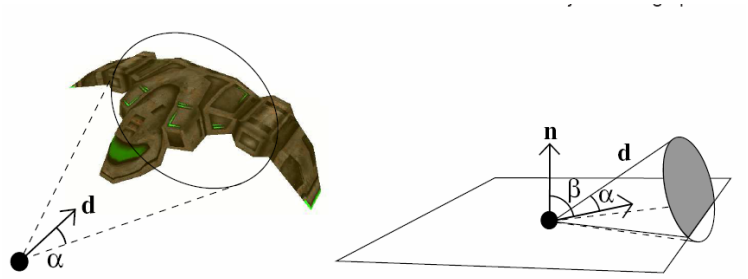
• render world space position and normals of all shadow receivers in the scene, including occluders.

• 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. The back faces of the occluder's grid were chosen, 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.

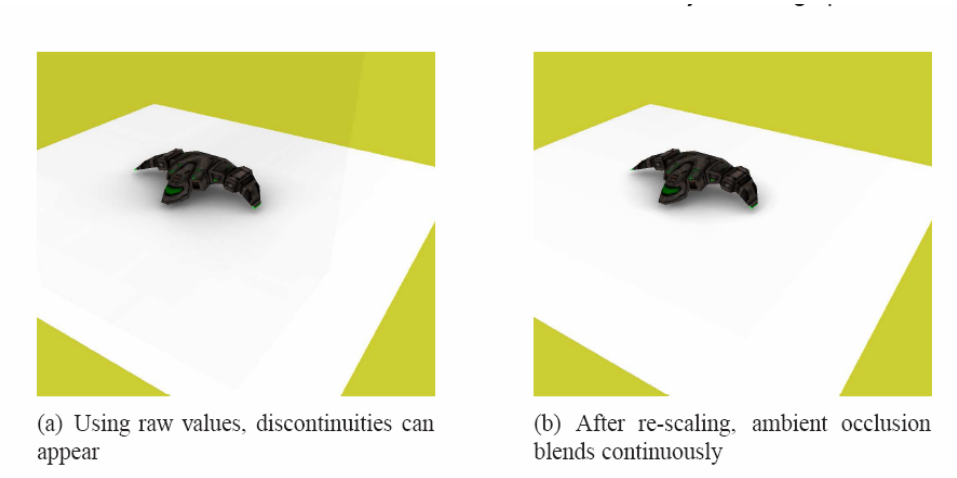
For more accurate ambient occlusion effects, the average occluded direction is also stored. That is equivalent to storing the set of occluded directions as a cone (see *Fig. 2.1.7*). The cone is defined by its axis (d) and the percentage of occlusion a (linked to its aperture angle α). Axis and percentage 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 d stored in the RGB-channels and occlusion value a stored in the A-channel.



(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.

Fig. 2.1.7. Ambient occlusion is stored as a cone.

When we there are several moving occluders in the scene, occlusion values from each moving occluder are computed, and merged together. The easiest method to do this is to use OpenGL blending operation. The occlusion value computed for the current occluder is blended to the color buffer, multiplicatively modulating it with (1 − a) as shown in the previous paper. Each occluder is rendered sequentially, using an 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.
An important parameter of the algorithm is the spatial extent of the grid. If the grid is too large, there is a risk of under-sampling the variations of ambient occlusion, otherwise the resolution has to be increased, thus increasing the memory cost. If the grid is too small, we would miss some of the effects of ambient occlusion. The optimal spatial extent of the grid is computed based on the bounding box of the occluder (see *Fig. 2.1.8*).



(a) Using raw values, discontinuities can appear

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

Fig. 2.1.8. We need to re-scale occlusion values inside the grid to avoid visible artifacts.

This is a nice method, in which the dynamic object problem is also solved. Its most important drawback is that it cannot handle the self-occlusion effect. Furthermore, it is limited to rigid body motion and is not suitable for deformable objects.

2.1.4. Real-time Ambient Occlusion for Dynamic Character Skins

The method described in this section, computes ambient occlusion values in real-time on dynamic character skins. The first method (in section 2.1.1) described an algorithm for calculating ambient occlusion in real-time with multiple passes of hardware, each one reducing the error in the approximation. In contrast, this method pushes much of the work to the precomputation stage, so evaluation consists of a fast, single pass. This method is designed to work with meshes that are deforming based on a low-dimensional set of parameters, as in character animation. The inputs are rendered ambient occlusion values at the vertices of a mesh deformed into various poses, along with the corresponding degrees of freedom of those poses. Character skins have the property that they are controlled by a low number of parameters, namely joint angles (one popular method for controlling character skins is linear blend skinning). While the skin changes significantly between poses, it does so in a well-defined parameter space. Furthermore, ambient occlusion on the mesh is a function of triangle position, which is in turn, a function of joint angles. Therefore, ambient occlusion at a point is expressed as a function of joint angles. Joint angles, however, do not provide a uniform space in which to compare poses, so instead of joint angles, poses are represented with handle positions. A handle is placed at the joint positions, plus a single handle per bone that is offset from the bone axis, (see *Fig. 2.1.9*). This defines a coordinate frame for each bone, and fully specifies the position and orientation of the bone in space. Pose is defined as be the vector of handle positions at a frame of motion data.

Given a set of ambient occlusion values and the corresponding poses, a set of linear functions is built that express ambient occlusion at a point in terms of pose. The training data consists of ambient occlusion values at the vertices of the skin, for a set of poses taken from a motion capture database. It is noted that ambient occlusion at a point on the mesh changes smoothly for small changes in pose. Therefore, if ambient occlusion is decomposed over the set of poses, this function can be modeled linearly. So, several linear functions are built, each one local to a different region of pose space. The domain of each locally linear approximation is a set of poses

that are close together. Because ambient occlusion changes smoothly as pose changes, it follows that poses close together should have similar ambient occlusion values. In fact, this method assumes that ambient occlusion and pose are linearly related, at least within local regions, and clustering techniques are used to find these local regions. The algorithm uses k-means clustering to group the degrees of freedom into a small number of pose clusters. Because the pose variation in a cluster is small, this method can define a low-dimensional pose representation using principal component analysis (PCA). Within each cluster, ambient occlusion is approximated as a linear function in the reduced-dimensional representation. The ambient occlusion at vertices in a local neighborhood tends to change similarly with changes in pose. Therefore, these vertices can be grouped together and store a single coefficient vector for the group. Note that this method computes pose clusters first, then the optional vertex clusters. This allows a different spatial compression within each pose cluster (see *Fig. 2.1.10*). When drawing the character, moving least squares are used to blend the reconstructed ambient occlusion values from a small number of pose clusters.
Still, this method requires a huge pre-computation step, where linear functions are built that express ambient occlusion at a point in terms of pose and where all the clustering is done.

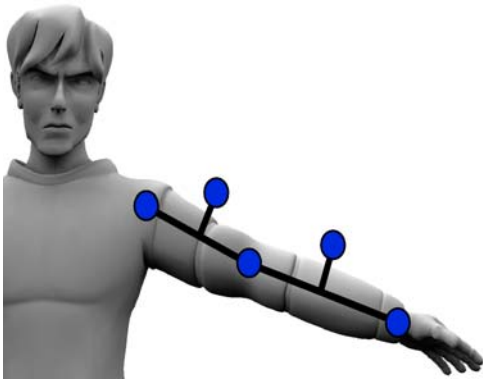


Fig. 2.1.9. Pose handles for the left arm are shown as blue circles.

Fig. 2.1.10. The blue patch on the shoulder shows a sample vertex cluster. This is one of 5000 clusters. The vertices in this cluster have similar ambient occlusion values over the pose cluster from which this frame was taken. Rather than storing a coefficient vector for each vertex, this method stores a single coefficient vector for the vertex cluster.

2.1.5. Hardware-accelerated Ambient Occlusion Techniques on GPUs

A multi-pass algorithm that separates the ambient occlusion problem into high-frequency, detailed ambient occlusion and low-frequency, distant ambient occlusion domains, both capable of running independently and in parallel, is presented in [Shan06].

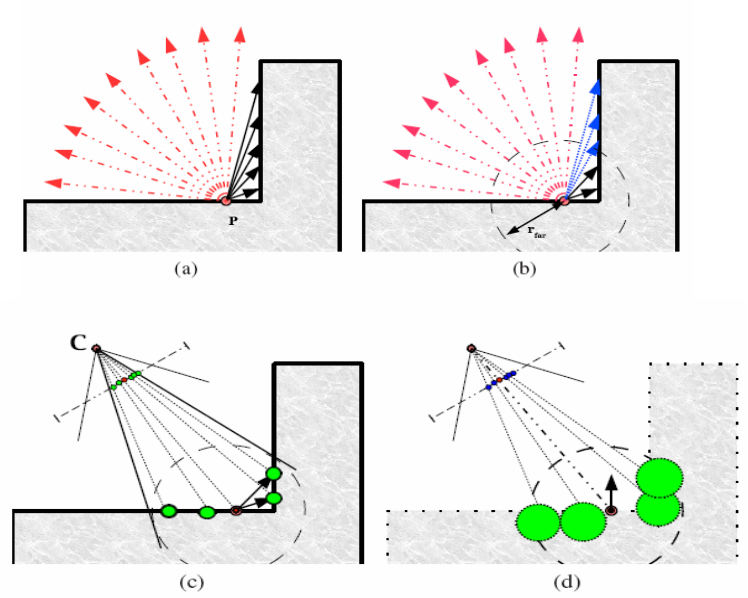


Fig. 2.1.11. Local ambient occlusion in image-space: (a) Rays emerging from a point P many of which are not occluded (marked red). (b) Rays being constrained within a distance of r_{far}, with distant occluders being neglected (blue arrows). (c) Pixels as viewed by a camera C. Neighboring pixels are obtained (marked in green). (d) these pixels are de-projected back to world-space and approximateed using spherical occluders (green circles). These are the final approximated occluders around the point P that are sought in image-space. Note that the spheres are pushed back a little along the opposite direction of the normal so as to prevent incorrect occlusion on flat surfaces.

The high-frequency detailed approach (see *Fig. 2.1.11*), uses an image-space method to approximate the ambient occlusion due to nearby occluders (caused by high surface detail). More specifically, it is noted that ambient occlusion needs to be calculated only for those receiver points that are visible to the camera, which correspond to the pixels that are present in the Z−buffer. Each pixel in the *Z−buffer* corresponds to a point in the world space, and given the normal buffer too, the position and normal for a given camera pixel can be obtained. Because each pixel in the ND−buffer (the combination of the two mentioned buffers) corresponds to a sample of some surface in the world-space, if that surface is approximated properly (with a sphere here), it can be used as an occluder to other objects.

Given the fact that the occluders that are nearby in world-space to a particular receiver point are also nearby to the corresponding projected pixel in the ND−buffer (ignoring the depth test), only nearby pixels are searched for nearby world-space occluders (see *Fig. 2.1.12*). For a given receiver pixel all the neighbouring pixels' approximated ambient occlusion values (approximated ambient occlusion values at a receiver point due to a sphere) are summed.

A different approach is used for low-frequency ambient occlusion (see overview in *Fig. 2.1.14*). First of all, a spherical approximation is used for the underlying surface geometry (see *Fig. 2.1.13*). Then, it is observed that as the distance increases the subtended surface area on the unit hemisphere decreases. So, beyond a certain distance, the ambient occlusion is less than a very small number, ε (ε cannot be visually distinguished from 0 ambient occlusion). This way exhaustive iteration through all occluders for all pixels is avoided.

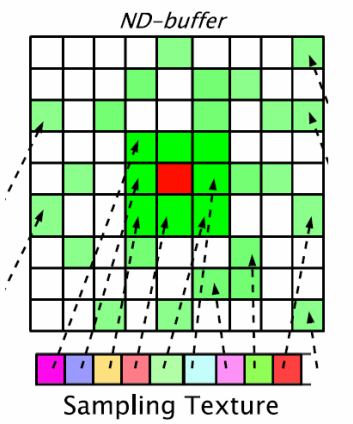


Fig. 2.1.12. This image shows a portion of the ND−buffer and the neighboring samples. We gather neighboring pixel samples around the receiver pixel p (shown in red) in the ND−buffer. The neighboring pixels (shown in green) are sampled randomly around the given receiver pixel so as to access as many pixels possible. The number of sample pixels to be gathered is calculated using the projected distance r_{far} at **p**; a pixel far away requires fewer number of samples than one that is nearer to the camera. The sampling texture provides the random distribution for the samples' locations.

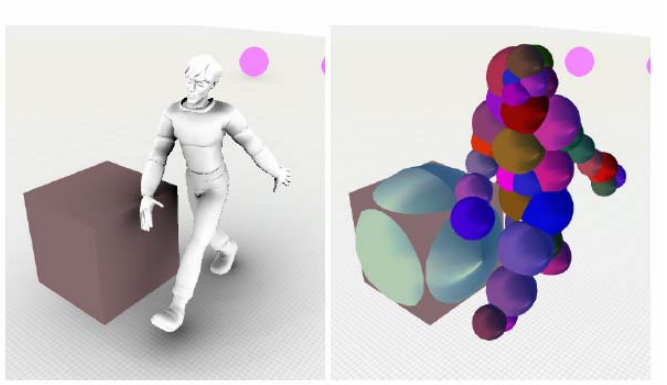


Fig. 2.1.13. The left image shows the distant occluder stage in isolation. Shadows and the image-space stage have been turned off to illustrate the nature of the distant occluder approach. The image on the right shows the spheres for the model that are being recalculated per frame. The rest pose provides us the initial position of these spheres due to Lloyd clustering, and we use this to reposition the spheres based on the new vertex positions. This recomputation is simple and fast.

It is also noted that for any two points that are at most d_{far} apart, their projected distance on the ND−buffer will not be farther than d_{far}. Thus, by covering all the pixels in the ND−buffer that are within a distance of d_{far} from $\mathbf{c}'$ (where $\mathbf{c}'$ is the projected pixel of the center of the sphere, **c**), we ensure that we cover all pixels that receive at least ε ambient occlusion due to the sphere <**c**, r>. For a given sphere, <**c**, r> , d_{far} is first calculated based on its radius, and then the fragment shader is invoked at all pixels q_i, that are at most d_{far} apart from $\mathbf{c}'$ in the image-plane, with the help of a billboard. The ambient occlusion effect due to a sphere <**c**, r> is "splatted" onto points that are capable of being influenced at least ε ambient occlusion. At every such invoked pixel, the position and normal are obtained, using the *ND−buffer* and an approximation for ambient occlusion is calculated (A_ψ). We then use the GPU's blending functionality to additively blend these A_ψ values.

Using the two algorithms mentioned above, two buffers, each containing the ambient occlusion approximation values for every pixel in the ND−buffer, are obtained, which are combined additively. The original color buffer is obtained and blended with the combined ambient occlusion value by multiplication (*Color* □ (1−*Occlusion*)). The case of the two approaches overlapping is avoided by setting the ambient occlusion value $A\psi = 1$, if the position **P** is within the sphere < **C**,r > in the case of the distant occluder approach.

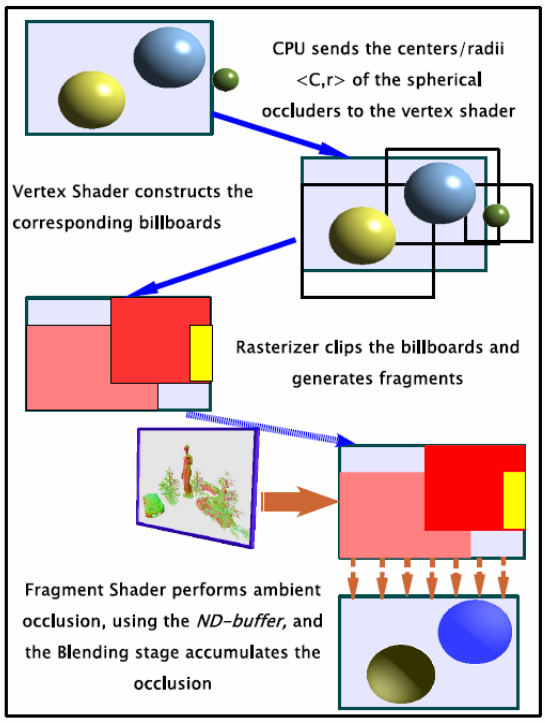


Fig. 2.1.14. Distant-occluder stage.

2.1.6. Presampled Visibility for Ambient Occlusion

The next method presented in [Gait08] uses a discretization approach. It accelerates the ray-object intersection test and in turn the computations of the visibility function of the lighting equation, by separating the task in two parts. First, at preprocessing time, a set of *displacement maps* is constructed (*Fig. 2.1.15*). These maps store the intersection distances of a hemisphere of rays originating from sample points on the bounding sphere of an object and directed towards the model itself. One map for each sample point is constructed (Algorithm 1). Then, at run time, when a ray from the environment towards an object intersects its bounding sphere, a simple ray-sphere intersection test is performed and the rest of the distance of the incoming ray at the given angle is recovered from the precomputed maps. The advantage of this method is that the bulk of the computation is moved to a pre-processing stage. The results are stored in compact grayscale textures (one byte per ray direction), providing for each object a constant size of additional information independent of the complexity of the original model. Then the real time algorithm performs a simple intersection test and a constant-time map lookup as in Algorithm 2.

This is a much faster technique, than all the previous methods presented, and a more general one, since it can also be used in ray tracing; the only problem is that due to its nature, it cannot calculate ambient occlusion for objects inside cavities, eg. it cannot calculate the ambient occlusion of surfaces inside a room (occluder).

Algorithm 1: Pseudo code of basic algorithm for displacement fields computation at preprocessing time.	**Algorithm 2: Pseudo code of basic algorithm for ambient occlusion rendering using displacement fields during real time processing**

Generate bounding sphere sample points
Generate samples of hemisphere of rays
for all bounding sphere sample points (u, v) **do**
  Align hemisphere of rays to normal at (u, v)
  for all rays (φ, θ) **do**
  if ray intersects the object **then**
  Normalize the distance (divide by 2 * R)
  Record distance in displacement map
  else
  Record distance in displacement map as 2*R
  end
end
	Generate hemisphere of ray samples
	for each occlusion receiver object **do**
	  for all points x on the occlusion receiver surface
	  do
	  for all emanating rays **do**
	  if ray intersects bound sphere of occluder obj.
	  Discretize intersection point (u, v)
	  Discretize ray (φ, θ)
	  Access distance in displacement map
	  end
	  Use distance for occlusion approximation
	  end
	  Compute occlusion at x
	  end
	end

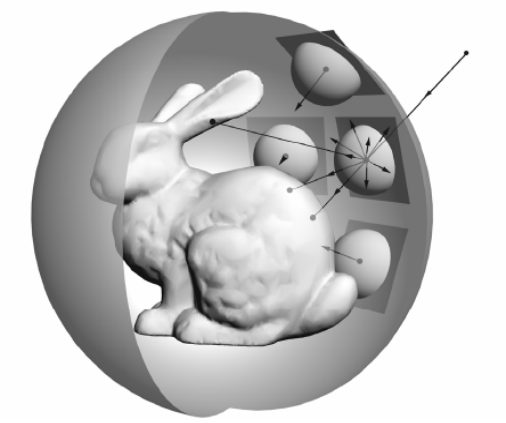


Fig. 2.1.15. A hemisphere of rays emanating from the bounding sphere towards the object is precomputed for a large number of sample points on the sphere.

2.1.7. Screen-space Ambient Occlusion

Screen-space Ambient Occlusion (SSAO) is a rendering technique for efficiently approximating the ambient occlusion effect in real-time, which was used for the first time in the 2007 PC game "Crysis". The algorithm is executed purely on the computer's GPU and implemented as a pixel shader, analyzing the scene depth buffer, which is stored in a texture. For every pixel on the

screen, the pixel shader samples the depth values around the current pixel and tries to compute amount of occlusion from each of sampled points. In its simplest implementation, the occlusion factor depends only on the depth difference between sampled point and current point. Without additional smart solutions, such a brute force method would require about 200 texture reads per pixel for good visual quality. This is not acceptable for real-time rendering on modern graphics hardware. In order to get high quality results with far fewer reads, sampling is performed using a randomly-rotated kernel. The kernel orientation is repeated every *N* screen pixels in order to have only high-frequency noise in the final picture. In the end this high frequency noise is greatly removed by a *N*x*N* post-process blurring step taking into account depth discontinuities (using methods such as comparing adjacent normals and depths). Such a solution allows a reduction in the number of depth samples per pixel to about 16 or less while maintaining a high quality result, and basically allows the use of SSAO in real-time applications like computer games. It is possible to use even only 2 samples, producing rather binary-looking noise, still looking like ambient occlusion. The result of this method is shown in *Fig. 2.1.16*.

Fig. 2.1.16. SSAO component of typical game scene

2.2. Voxelization

Voxelization (or volume synthesis or 3D scan-conversion) is the process of converting a geometric representation of a synthetic model into a set of voxels that "best" represents that synthetic model within the discrete voxel space. In this section we shall present two of the most popular voxelization algorithms; the depth-buffer-based voxelization algorithm [Kara98], which is a very fast voxelization algorithm, that does not work for objects with internal cavities, and the slicing algorithm [Chen97], which is fast algorithm (not as fast as the first one though), that works for all kinds of objects, if their models are specially constructed.

2.2.1. Depth-buffer-based Hardware Accelerated Voxelization

The first voxelization algorithm that we will examine [Kara99] is a very fast algorithm which is based on the creation of volume data using depth information from 6 different views of the object and could be regarded as an application of the z-buffer. The object is surrounded by three mutually perpendicular pairs of z-buffers ([*x1*,*x2*], [*y1*,*y2*], [*z1*,*z2*]), aligned with the three primary axes of the object or the WCS. Each pair consists of two buffers, perpendicular to the same axis but in opposite sides of the object, in such a way that they are facing each other (*Fig. 2.2.1*). Each pair is holding depth information for the object from a different viewing angle. Voxelization is accomplished by scanning the volume space and checking whether each voxel is between the limits imposed by the buffers. More specifically, voxelization, i.e. deciding whether a voxel belongs to the object (interior or contour) or not, is equivalent to examining if the voxel lies within the boundaries defined by the corresponding depth values of the buffers (*Fig. 2.2.2b*). Since voxel coordinates are defined relative to the voxel cube size, while z-buffer values lie in an implementation dependant range, the position of *v(i,j,k)* voxel needs to be converted to the z-buffer range for the comparison.

This is a very simple, easy to implement, yet very fast method, which is constrained by certain limitations; the most important, is that it is not able to correctly reconstruct objects with internal cavities or other parts, which can not be seen from the chosen viewing angles.

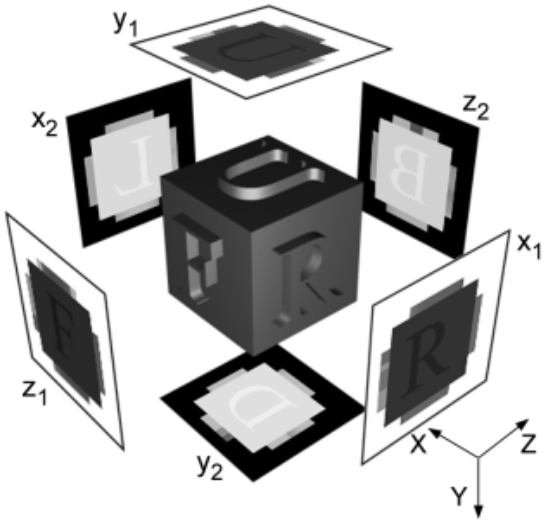


Fig. 2.2.1. Depth buffer setup for the voxelization.

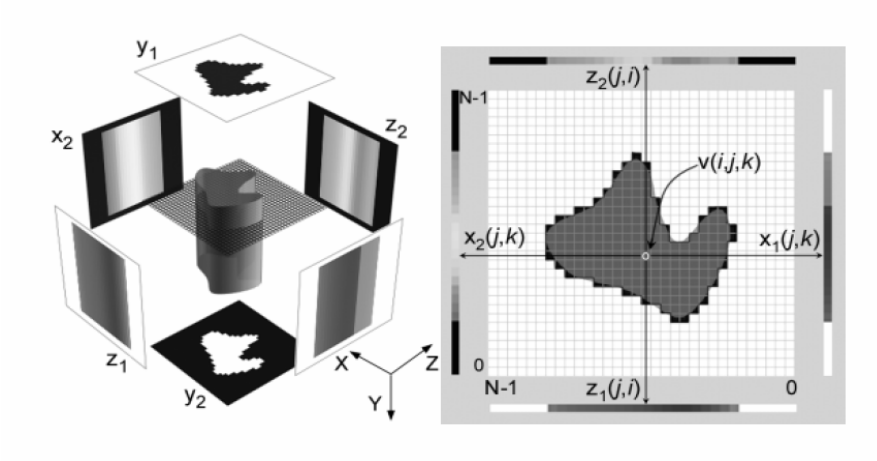


Fig. 2.2.2. The principle of the depth-buffer-based voxelization algorithm.

2.2.2. Volume-slice Hardware Accelerated Voxelization

The second algorithm presented here, [Chen98], is not as fast as the first one, but it can correctly reconstruct objects with internal cavities, provided that the surfaces are well defined and closed. The algorithm is based on the fact that surface graphics displays a curve or a surface by a 2D scan conversion (or rasterization) process. When only a slice of the object is displayed, the result is essentially a slice of the volume from a 3D scan conversion. Since 2D scan conversion is implemented in hardware in modern graphics systems, 3D voxelization ought to be able to take advantages of it for better performance.

A bounding box is first defined over the scene as the volume space for voxelization. The algorithm proceeds by moving a near and a far cutting plane perpendicular to the Z-axis (Z-planes), parallel to the projection plane, with a constant step size in a front-to-back order. The thin space between two adjacent Z-planes within the volume space is called a slice (as shown in *Fig. 2.2.3*). For each new Z-plane, the algorithm defines the new slice as the current orthogonal viewing volume, and renders all the curve and surface primitives using standard OpenGL procedures. Since the boundary planes of the viewing volume are considered clipping planes in OpenGL, the clipping mechanism of the graphics engine will ensure that only the parts of the curves or surfaces within the thin viewing volume are displayed. The resulting frame buffer image from the display of this slice becomes one slice of the voxelization result of the 3D scene. When the algorithm moves from slice to slice, it basically generates the slices of the volume representation in a front-to-back order. Each such slice, once generated in the frame buffer, can be written to a 3D texture memory as one slice of the a texture volume, which can be immediately rendered by 3D texture mapping, or later written, as a whole volume, back to the main memory. It is noted that the voxelization result should always be rendered and examined in the 3D texture memory first before being written to the main memory. The Z-distance between adjacent Z-planes determines the Z-resolution of the volume representation. The resolutions in the X and Y directions are defined by the size of the display window.

The object's interior voxels, which also need to be filled, are however not explicitly scanned by this process. To generate the interior voxels, the algorithm employs the frame buffer blending function feature in OpenGL with a logical XOR operation to carry the boundary information to the interior of the solid object. This approach is based on the principle that when shooting a ray from a pixel to the object space, the entering points and the exiting points on the ray against a well-defined solid object always appear in pairs, and the voxels between each pair of entering and

exiting points are the interior voxels. The XOR operation also ensures the voxel coherency in the Z-direction for polygons that tend to be perpendicular to the viewing plane.

Still this algorithm is not good for all models, since it is not always sure that the model's boundary surfaces do not intersect each other. If the latter is the case, the algorithm cannot be used to correctly reconstruct objects. As will be discussed in Section 3.1.2, this is why we have opted for a custom slice-based voxelization algorithm that dispenses with the XOR operation (the internal voxels are not needed). To cover for the incremental scan-conversion discrepancy of the polygon slope, the voxelization is performed thrice, using slices perpendicular to all three primar axes.

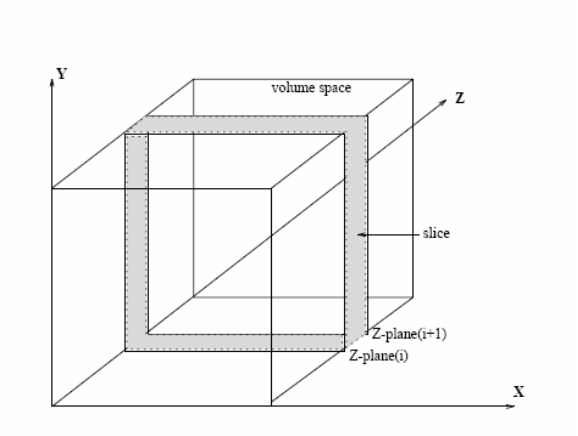


Fig. 2.2.3. Volume Slicing

3. Description of the Method

In this section we shall present a novel real-time algorithm for pre-calculated visibility, that can be used to produce a visually pleasant ambient occlusion and lighting approximation.

Unlike other methods that store directional visibility information for every location in space [Gait08], [Kont05], [Malm06], it does not use directional data, so it is a very fast method that uses a comparatively small amount of memory storage. Furthermore, it does not suffer from view-dependency, nor has it the local effect depth-map-based ambient occlusion algorithms have [Shan06]. This method, unlike others, is also capable of illuminating appropriately the internal part of objects (e.g. the inside of a room) ([Gait08], [Malm06]), and it can handle self-occlusion ([Malm06]). The algorithm presented can handle active and passive illumination effects uniformly, which is something that none of the other algorithms presented can do.

The main disadvantage of the algorithm that will be presented is that it cannot handle deformable objects, unlike the methods presented in [Shan06] and [Kirk07]

The main idea of the method presented, can be summed up in two general algorithms, one for the pre-processing, and one for the real-time stage. Algorithm 3.1 describes the pre-processing stage of the method, while the Algorithm 3.2 explains the calculation stages for self-occlusion, which in turn will be extended in Section 3.3 to cover both self and inter-object occlusion:

Algorithm 3.1 – pre-processing stage

```
for every object o
   Create a bounding box around o (3.1.1)
   Voxelize o and mark as "internal" all voxels intersected by o's surface (3.1.2)
   for every voxel v
      Calculate a proximity value to v's neighboring surfaces (3.1.3)
   end
   Store the values in a 3D texture
end
```

Algorithm 3.2 – real-time stage

```
for every (receiving) object o
   for every fragment (fragment shader)
      Take a number of proximity samples centered along the normal direction, within distance r
      (3.2)
      Reconstruct a surface occlusion value based on these samples (3.2)
End
```

More specifically, first each object in the scene is voxelized, so a voxel grid is created around the object and the voxels intersected by its surface are marked as "internal" (maximum surface proximity – minimum distance).

Then, for every voxel center, an approximate ambient occlusion value is calculated and stored in the voxel grid; this value is position-dependent – not directional, so that when, in real-time, a surface point **p** needs to be shaded using the ambient occlusion effect, its occlusion value is derived from the positional samples already stored in the proximity volume texture. By default, we cannot calculate a point's "obscurance" or ambient occlusion, unless it belongs to a surface. In our case, the center of each voxel does not belong to any surface, so a new term has to be defined,

Positional Ambient Occlusion, which represents the proximity of all surrounding surfaces (up to a maximum distance) to that point.

As will be discussed later, an interesting consequence of the proximity value storage used in our algorithm is that it is possible to incorporate in the same ambient illumination calculation the contribution of direct diffuse illumination, such as light emitters of various sizes, shapes and light emission distributions.

To calculate the positional ambient occlusion for every voxel center, a flooding algorithm is used; in this algorithm, the currently calculated proximity value of every voxel is diffused to its neighbors via an iterative propagation scheme, so that each voxel eventually stores an approximate proximity value or, in other words, cumulative distance to the neighboring surfaces. This approach is faster than explicitly calculating the cumulative distance for each individual voxel by a Monte-Carlo method or any other used in Ambient Occlusion. The proximity values are stored in a 3D texture, and are passed on to a shader during real-time rendering.

In the shader, for every fragment, the proximity values are sampled along the normal direction from the stored distance information, and based on that, an occlusion value is reconstructed. The proximity volumes are expressed in the local space of each object and during an animated sequence with multiple objects, the volumes are also transformed along with their associated objects. In order to account for the intra- and inter-object ambient shading, the cumulative contribution of the proximity volumes of all nearby objects is estimated.

Each one of these algorithms will be presented in full detail in the next subsections. More specifically, as it can be seen above, next to each algorithm's non-trivial step, in parentheses is mentioned the exact subsection where that step is presented.

But before we see the algorithm in full detail, we should first derive a closed formula for the Positional Ambient Occlusion.

3.1. Positional Ambient Occlusion

In our case, for every voxel center, an approximate ambient occlusion value will be calculated and stored in the voxel grid; this value will be position-dependent – not directional , so that when, in real-time, a surface point **p** needs to be shaded using the ambient occlusion effect, its occlusion value will be derived from the positional samples already stored in the proximity volume texture. Using the standard ambient occlusion formula, we cannot calculate a point's "obscurance" or ambient occlusion, unless it belongs to a surface; here, the center of each voxel does not belong to any surface. So, we need to introduce a new term, the "Positional Ambient Occlusion".

But before we get to that, we should see how the classic ambient occlusion formula (*eq. 1.1.1*) is calculated.

Ambient Occlusion on a point **x** of a surface with normal $\vec{\mathbf{n}}$ is calculated by integrating a visibility function, over a hemisphere Ω (for all directions $\vec{\omega}$) around **x**. If *dA* a very small patch where we want to calculate ambient occlusion, $\vec{\omega}$ the direction in the hemisphere where we are currently looking, $\vec{\omega}_{proj}$ the projected solid angle of $\vec{\omega}$ and $\rho(d(\mathbf{x},\vec{\omega}))$ the obscurance caused by the interference of a surface at distance $d(\mathbf{x},\vec{\omega})$ from **x** then:

$$AO(x,\vec{n}) = \int_{\Omega} V(dA,\vec{\omega}) d\vec{\omega}_{proj} =$$
$$= \int_{\Omega} \rho(d(\mathbf{x},\vec{\omega})) \cdot \cos\theta d\vec{\omega}_{proj} =$$
$$= \int_{\Omega} \rho(d(\mathbf{x},\vec{\omega})) \cdot \cos^2\theta d\vec{\omega} =$$
$$= \int_{0}^{2\pi}\int_{0}^{\frac{\pi}{2}} \rho(d(\mathbf{x},\vec{\omega})) \cdot \cos^2\theta \cdot \sin\theta d\theta d\phi =$$
$$= 2\pi \int_{0}^{\frac{\pi}{2}} \rho(d(\mathbf{x},\vec{\omega})) \cdot \cos^2\theta \cdot \sin\theta d\theta$$

If we want to find the value of maximum ambient occlusion, i.e. the occlusion for maximum obscurance ($\rho(d(\mathbf{x},\vec{\omega}))_{\max}=1$), the above becomes:

$$AO(x,\vec{n}) = 2\pi \cdot \int_{0}^{\frac{\pi}{2}} \cos^2\theta \cdot \sin\theta d\theta = \pi$$

The ambient occlusion value should be between 0 and 1, so we normalize it by dividing it by π; this is how the formula for the *normalized* ambient occlusion is derived (*eq. 1.1.1*).

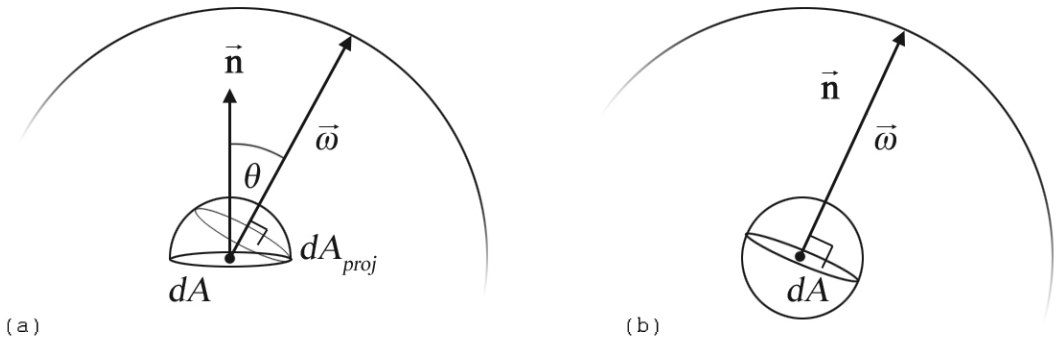


Fig. 3.1.1. (a) Ambient occlusion calculation, (b) Positional Ambient Occlusion calculation

Now, in order to define the Positional Ambient Occlusion, we consider a sphere with an infinitesimal diameter centered at the point where we want to calculate it. We need to calculate the percentage of blocked incident light on the sphere. Illumination – or rather here, the absence thereof - reaches the volume of the sphere centered at a voxel's center from every direction. Since we have a sphere, the projected area in the incident direction $\vec{\omega}$ is always a full disk ($\vec{\omega}_{proj} = \vec{\omega}$).

Therefore the incident flux is always maximum, because the angle between the sample direction $\vec{\omega}$ and the surface's normal $\vec{n}$ is 0 (see *Fig. 3.1.1b*), thus eliminating the cosine term in the ambient occlusion equation:

$$C(x) = \int_{\Omega} V(dA, \vec{\omega}) d\vec{\omega}_{proj} =$$
$$= \int_{\Omega} \rho(d(\mathbf{x}, \vec{\omega})) \cdot \cos\theta d\vec{\omega}_{proj} =$$
$$= \int_{\Omega} \rho(d(\mathbf{x}, \vec{\omega})) \cdot \cos\theta d\vec{\omega} =$$
$$= \int_{\Omega} \rho(d(\mathbf{x}, \vec{\omega})) \cdot \cos 0 d\vec{\omega} =$$
$$= \int_{\Omega} \rho(d(\mathbf{x}, \vec{\omega})) d\vec{\omega}$$

If we want to find the value of positional ambient occlusion for the maximum obscurance ($\rho(d(\mathbf{x}, \vec{\omega}))_{max}$=1), the above becomes:

$$C(x) = \int_{\Omega} d\vec{\omega} = 4\pi$$

The positional ambient occlusion value (similar to the conventional ambient occlusion) should be between 0 and 1, so we normalize it by dividing it by 4π.

So, the type for *Positional Ambient Occlusion* becomes after normalization:

$$C(x) = \frac{1}{4\pi} \int_{\Omega} \rho(d(\mathbf{x}, \vec{\omega})) d\vec{\omega} \qquad (3.1.1)$$

3.2. Pre-processing stage

In the pre-processing stage, first of all, a bounding box is created for every object in the scene, which leaves some vacant space in order for the positional ambient occlusion to be calculated around the geometry. Then, every object is voxelized and all voxels intersected by its surface are marked appropriately. The next and most important part of this stage, is the calculation of a positional ambient occlusion (proximity) value for every voxel around every object. Finally, all the calculated positional ambient occlusion values are stored in a 3D texture, which is passed to a shader, utilized in the real-time part.

3.2.1. Creation of Bounding Box

For every object in the scene, a bounding box is created around it. As the method requires the generation of proximity values within a distance r_P (we shall call this distance *propagation radius* from now on) from the objects' surface, the bounding box should not be tightly constructed around it but rather extend an extra distance r_P on every side (*Fig. 3.2.1.1*). r_P is estimated based on the overall size of the scene. One strategy to do this, if the independent objects in the scene

roughly move inside a large object (e.g. inside a game level map), is to take r_P as a fraction of the diagonal of the bounding box diagonal of the union of objects. Another approach is to construct a maximal bounding box, i.e. a bounding box whose sides are the largest sides of all individual bounding boxes and regard a fraction of its diagonal as r_P. In our tests, this fraction is 10-15%.

r_P determines the maximum distance that the proximity values calculated will be propagated, and consequently, it will contribute to the determination of how far the object's shadowing effect reaches.

The size of an object's bounding box determines the dimensions of the associated voxel space that will be used to store the proximity values. More specifically, the voxel space size is proportional to the ratio between the bounding box size and that of the reference bounding box, which, depending on the approach for the determination of r_P, is either the bounding box of the union of objects or the maximal bounding box. In the latter case, and assuming that we require power-of-2 voxel space dimensions, the voxel space for each object is estimated as shown in the code fragment Code 3.2.1.1.

Code 3.2.1.1. Voxel space estimation

```
if ( document.data_initialized )
            return;

if (document.num_models==0)
{
    document.data_initialized=true;
    return;
}
int i, dim, tmp_dim;

// find max dim
Vector3D dd = document.models[0]->BoundBox.max-
              document.models[0]-BoundBox.min;
DBL maxsize = dd.x;
DBL ratio;
if (dd.y>maxsize)
    maxsize = dd.y;
if (dd.z>maxsize)
    maxsize = dd.z;
for (i=1; i<document.num_models;i++)
{
   dd = document.models[i]->BoundBox.max-document.models[i]->BoundBox.min;
   if (dd.x>maxsize)
      maxsize = dd.x;
   if (dd.y>maxsize)
      maxsize = dd.y;
   if (dd.z>maxsize)
      maxsize = dd.z;
}

// find closest power of 2 dimension according to relative model scale.
for (i=0; i<document.num_models;i++)
{
    //calculate dimensions
    dd = document.models[i]->BoundBox.max-document.models[i]>BoundBox.min;
    ratio = dd.x/maxsize;
```

```
    dim = document.volume_res;
    while ( (fabs(ratio-dim/(float)document.volume_res) >
             fabs(ratio-dim/(float)(2*document.volume_res)) ) && dim>32)
         dim/=2;
    ratio = dd.y/maxsize;
    tmp_dim = document.volume_res;
    while ( (fabs(ratio-tmp_dim/(float)document.volume_res) >
             fabs(ratio-tmp_dim/(float)(2*document.volume_res)))
             &&tmp_dim>32 )
         tmp_dim/=2;
    if (tmp_dim > dim)
         dim = tmp_dim;
    ratio = dd.z/maxsize;
    tmp_dim = document.volume_res;
    while ( (fabs(ratio-tmp_dim/(float)document.volume_res) >
             fabs(ratio-tmp_dim/(float)(2*document.volume_res) ) )
             && tmp_dim>32 )
         tmp_dim/=2;
    if (tmp_dim > dim)
         dim = tmp_dim;
    document.models[i]->volumeInit(dim);
}
```

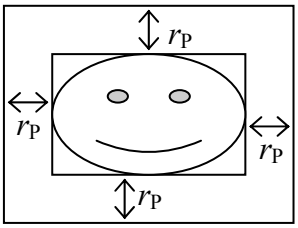


Fig. 3.2.1.1. The extended volume. Initial bounding box / Radius r_p relation

3.2.2. Voxelization

As we have mentioned earlier in Section 2.2, none of the existing fast voxelization algorithms is good enough when we can make no assumption on the quality and shape of the meshes. It is important for real-time applications to impose no restrictions on the data used as it is more convenient in terms of modeling to allow crossing or open (non-watertight) surfaces. Since voxelization is done in the pre-processing stage, a very fast algorithm is not required.

We use a variant of Chen's algorithm [Chen98]. More specifically, the algorithm follows the same steps as Chen's algorithm, as far as the slicing part is concerned. The main difference is that it doesn't use the XOR operation (because not all models are perfect), so the volumes are actually hollow, but since we calculate proximity this will not affect neither the calculations, nor the visual result.

Another problem presents itself, due to the fact that we do not use the XOR operation. Scanline algorithms are incremental by nature and require that the slope of the dependent variable on the increment is less than 1. As far as the scan conversion of a polygon onto the view plane is concerned, there are no holes generated but when the fragments are stored as voxels, discontinuity in the Z-axis is catastrophic. The XOR operation indirectly solved this problem (by

filling in the missing fragments) but in our case we need to find another solution. The problem is solved by repeating the scan-conversion process 3 times, one for each primary direction (by rotating the object 2 times). This way, we ensure that the depth-discontinuity in one orientation of the view plane will be remedied in one of the two others.

For every slice, a procedure that is exactly the same as the 2D scan conversion algorithm is called, in order to voxelize the triangles on that slice. Three temporary matrices are needed to store the temporary volume results of the three slicing procedures (one for each different orientation of the object as explained above). Finally, those three volumes are combined into one, using the bitwise OR operation.

When a voxel is occupied, the value 255 is stored in it, and the value 0 is stored when it is not. So the algorithm used is as shown in Algorithm 3.1.2.1:

Algorithm 3.2.2.1 – Voxelization Algorithm

```
zplane1 = Z value of the front face of the bounding box;
zstep = bounding_box_size / N;
for i = 1 to N                          // Slicing loop
   zplane2 = zplane1 + zstep;
   Define the thin orthogonal viewing volume;
   Execute the 2D scan conversion algorithm for all object faces
   Copy the result of the scan conversion in the respective slice of a temporary volume matrix, vMatZ
   zplane1 = zplane2
end

Rotate all points 90° around the Y axis   // Same as scan-converting from a different direction
xplane1 = X value of the front face of the bounding box;
xstep = bounding_box_size / N;
for i = 1 to N                          // Slicing loop
   xplane2 = xplane1 + xstep;
   Define the thin orthogonal viewing volume;
   Execute the 2D scan conversion algorithm for all object faces
   Copy the result of the scan conversion in the respective slice of a temporary volume matrix, vMatX
   xplane1 = xplane2
end

Rotate all points -90° around the Y axis
Rotate all points 90° around the X axis
yplane1 = Y value of the front face of the bounding box;
ystep = bounding_box_size / N;
for i = 1 to N                          // Slicing loop
   yplane2 = yplane1 + ystep;
   Define the thin orthogonal viewing volume;
   Execute the 2D scan conversion algorithm for all object faces
   Copy the result of the scan conversion in the respective slice of a temporary volume matrix, vMatY
   yplane1 = yplane2
end

Combine the 3 temporary volumes, vMatZ, vMatX, vMatY, in one final volume using the OR
operation.
```

3.2.3. Proximity calculation

At this step, we already have a voxel space, with all occupied voxels marked with the value 255, from the previous step. Here, we want to calculate the positional ambient occlusion (proximity) at the center of every voxel. (see *Fig. 3.2.3.1*).

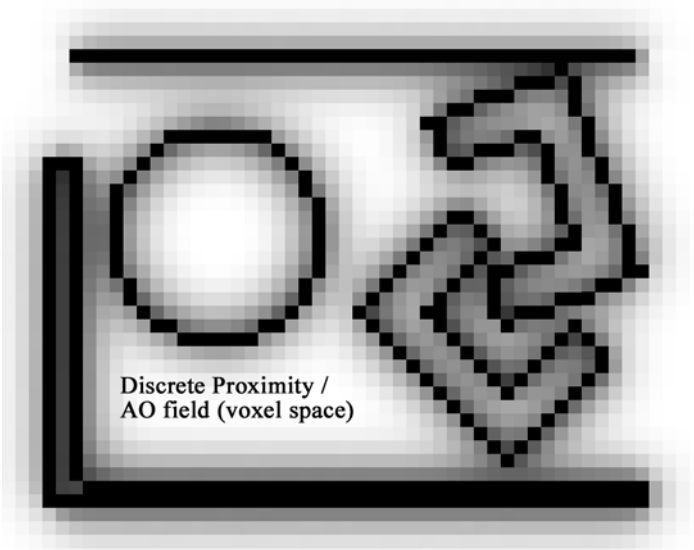


Fig. 3.2.3.1. After the voxelization step, the occupied voxels are marked with the value 255 (the black voxels in the picture). After the proximity calculation, a proximity (positional ambient occlusion) value will be calculated for every voxel (all the gray and white voxels).

One way to precompute the positional ambient occlusion is to explicitly sample the surrounding sphere of the voxel center, using the typical Monte Carlo ray casting method for every voxel's center, combined with the formula 3.1.1 for positional ambient occlusion. A much faster approach, especially for large models, is presented below.

We propose a flooding algorithm, according to which the proximity value of every voxel is diffused to its neighbors via an iterative propagation scheme, so that each voxel eventually stores an approximate value of positional ambient occlusion for all the (or proximity or cumulative distance to) the neighboring surfaces.

At every iteration of the algorithm, the proximity information is propagated outwards from the surface, one voxel at a time (hence the voxel's diagonal in world space units), up to a maximum number of iterations N_P directly dependent on the propagation radius r_P:

$$N_P = r_P / d_{voxel}$$

where the propagation radius is the one mentioned in 3.1.1.

At every iteration of the main loop, every voxel (i,j,k) is checked and if it is not occupied (its value is not 255), the average proximity, P_{ave}, of its all 26 neighboring voxels is calculated and then its proximity value, $P_{i,j,k}$, is set to $(P_{i,j,k} + P_{ave})/2$.

A temporary volume buffer is used to store step $(n+1)$ proximity values, in order not to interfere with the n-step's. The procedure is presented in the following algorithm listing.

Algorithm 3.1.3.1 – Proximity Calculation Algorithm

```
while iteration_num <  (propagation radius)/( voxel's diagonal)
   iteration_num++
   for every voxel
      if it is marked as "occupied" (value 255)
         copy the value to the temporary buffer
      else
         calculate the average proximity P_ave of all 26 neighboring voxels
         set current pixel's proximity value P to (P + P_ave )/2  in the temporary volume buffer
      end
   end
   copy the temporary buffer to the main volume buffer
end
```

In *Fig. 3.2.3.2*, a graph is presented of the proximity values against the distance from occupied voxels (in voxel units). The dotted thin line is the case where we have only one surface. The thin line is in the case we have another surface 15 voxels away from the first one. The bold line is the cumulative effect in the case we have both surfaces present. As the value of a voxel is a linear combination of its neighboring values, in this simple one-dimensional case it is quite obvious that the bold line is the sum of the other two lines. This will be a useful property of the discrete proximity field when later on we try to estimate the effect of multiple occluders to the ambient occlusion of a surface point.

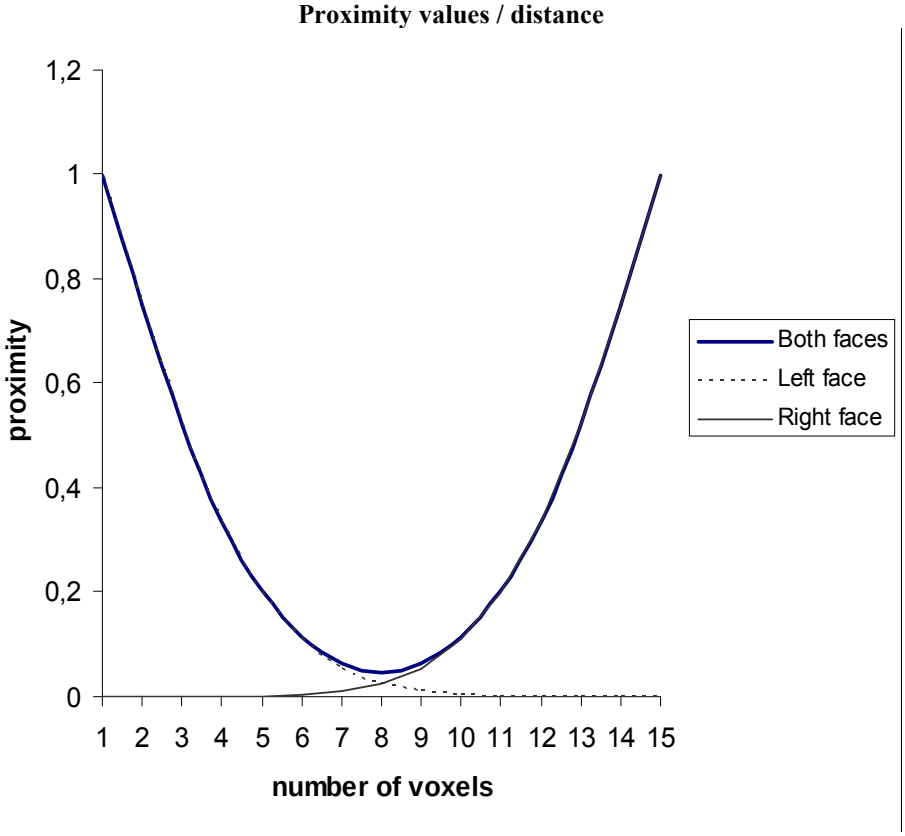


Fig. 3.2.3.2. Proximity values for two parallel surfaces. When two surfaces are present in the same proximity calculation (the two thin lines), the cumulative effect (bold line) is the sum of the proximity calculations of the two surfaces taken independently.

3.3. Real-time Ambient Occlusion Reconstruction

In this stage, a shader is used to illuminate each fragment. The fragment shader reads the proximity values calculated in the pre-processing stage as a 3D texture. For every fragment and its associated location in space **p**, it draws proximity samples from the 3D texture in a solid angle around the normal vector $\vec{\mathbf{n}}$ at **p** and, based on these, an occlusion value is reconstructed.

In order to reconstruct the directional ambient occlusion at **p**, for an ambient occlusion range *R,* the range is split into *N* equally sized segments. *N* is a predefined constant that as will be discussed in the test cases section, affects the quality of the final result and is also associated with the resolution of the voxel space.

It would seem natural to estimate the surface ambient occlusion of **p** only from the voxels near **p**. But as $C(\mathbf{x})$ is omnidirectional, there is a high probability that for $\mathbf{x} \approx \mathbf{p}$ occlusion is caused by the surface area near **p**. To this effect, we discard all samples whose proximity values decrease, as a function of distance to **p** (see *Fig. 3.2.1* and *Fig. 3.2.2*).

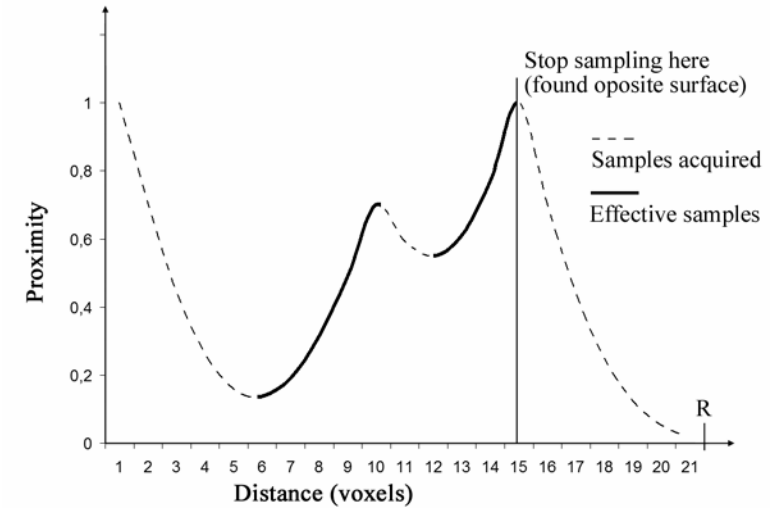


Fig. 3.2.1. Proximity (or positional ambient occlusion) – distance graph. Dotted line: samples taken are discarded. Normal lines: samples taken are used in the ambient occlusion reconstruction. When proximity values are decreasing, the samples taken are discarded. Sampling ends when proximity becomes 1, because in that case a surface is reached

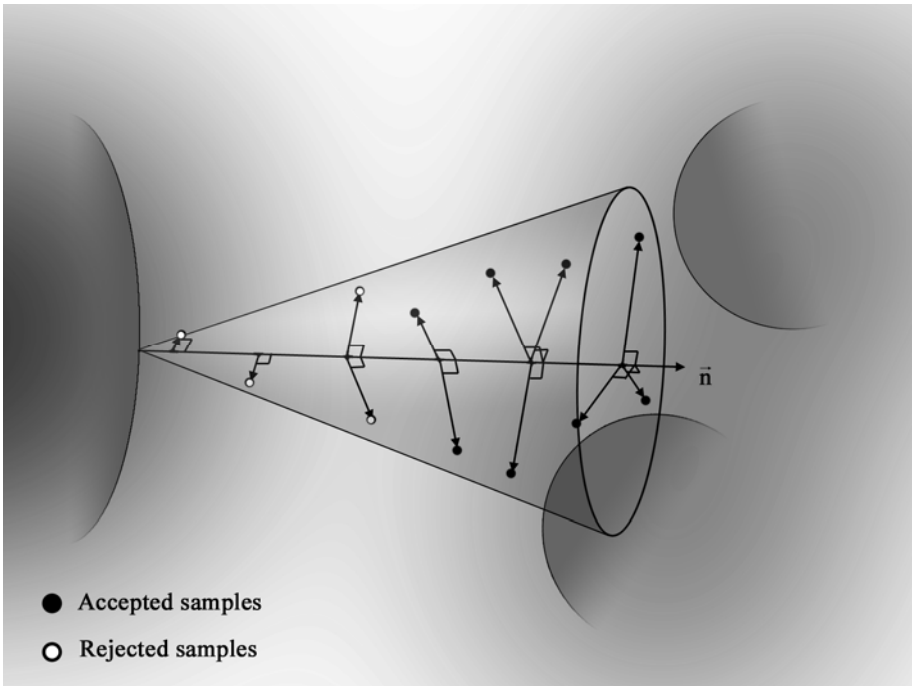


Fig. 3.2.2. Sampling strategy. All samples whose proximity values decrease, as a function of distance to **p** are discarded (white samples). We can also see that the longer the distance from **p**, the more samples need to be drawn from a greater surface.

The farther a sample **x** from **p**, the higher the probability that $\|\mathbf{x}-\mathbf{p}\|$ is comparable to the recorded proximity values, thus making $C(\mathbf{x})$ unreliable. This means that the longer the distance from **p**, the more samples need to be drawn in order to counter this uncertainty.

When at a sampling step the average proximity exceeds the maximum value limit (close to texture value 1) the sampling is terminated as further iterations would certainly produce samples that penetrate an opposite surface.

The step made in every iteration of the sampling along the normal direction is $\frac{R}{N}$.

The ambient occlusion effect range R is different from the propagation radius, mentioned in 3.1.1, because the latter shows haw far the proximity information is propagated in the pre-processing stage, while this radius shows the distance the proximity information calculated is taken into account in order to reconstruct the occlusion value. It is pointless for this radius to be greater than the propagation one, because, if it were, that would mean that the algorithm would look for proximity information in greater distance than the one in which it had calculated the proximity values it needed.

In the code segment *Code 3.3.1* the fragment shader code responsible for the real-time procedure described above, is presented.

Code 3.3.1 The fragment shader code responsible for the real time part of the method.

```
int i,j;
vec4 p4 = vec4(p,1);
vec4 prev_ao = texture3D(tex_occlusion, gl_TextureMatrix[1]*p4);
vec4 cur_ao;
vec4 dev_p;
vec3 step = (R/Nsamples)*n;
ao=vec4(0,0,0,1);
vec4 zero = vec4(0,0,0,0);
vec4 one  = vec4(1,1,1,1);
p4 += vec4(step,0);
int hits = 0;
for(i=0; i<Nsamples; i++)
{
   p4 += vec4(step,0);
   cur_ao = zero;
   for(j=0; j<i/2; j++)
   {
          PerturbPoint( dev_p, p4, n, (R*i)/Nsamples, R, j );
          vec4 sampleCoord = gl_TextureMatrix[1]*dev_p;
          if (any(lessThan(sampleCoord,zero)))
                continue;
          if (any(greaterThan(sampleCoord,one)))
                continue;
          cur_ao += texture3D(tex_occlusion, sampleCoord);
   }
   if(cur_ao[0]>=prev_ao[0])
   {
        hits++;
        ao+= cur_ao*(1 - i/Nsamples);
        if (cur_ao[0]>0.95)
        {
              break;
        }
   }
   prev_ao = cur_ao;
}
ao*=0.8;
ao = clamp(ao/Nsamples,0,1);
ao*=0.8;
ao.a = 1;
```

3.4. Inter–object Shading Effect

The algorithm described up to now shades one object in the scene by sampling appropriately the proximity (positional ambient occlusion) values calculated in the voxel space volume around it and reconstructing the ambient occlusion value for every fragment. This procedure alone is correct, when the scene consists of only one single object. If many objects are present, to obtain a realistic the above is obviously not enough, since objects must shade each other.

The calculation of the positional ambient occlusion in a volume around each object can be repeated independently for every object in the scene. To apply ambient occlusion to the whole scene, every object in it should be shaded correctly, also taking into account the proximity values calculated from other objects close to it.

Therefore, to shade an object (A), we need to transform its fragments to every other object's (B) texture space and apply the shader to them using the volume of B to draw positional ambient occlusion samples from. This is done for all occluding objects in the scene and finally, the results are combined additively.

To transform an object (A) to another object's (B) texture space, if $\mathbf{M}_A$ and $\mathbf{M}_B$ the transformations of objects A and B (see *Fig. 3.4.1*), respectively with respect to the WCS, we first transform A to B's local coordinate system, using the transformation $\mathbf{M}_B\mathbf{M}^{-1}_A$, and then we transform it to B's texture space, using the transformation $\mathbf{T}_{0.5,\,0.5,\,0.5}\,\mathbf{S}_{1/\text{volume B's size}}\mathbf{T}_{-\text{volume B's center}}$.

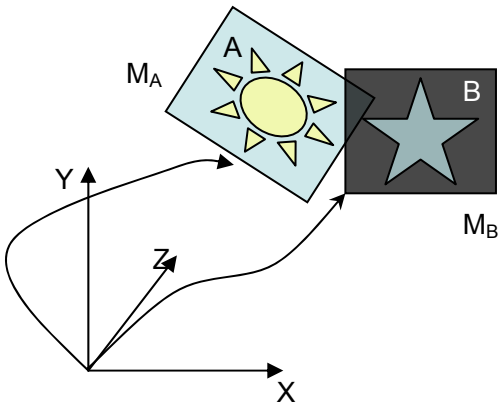


Fig. 3.4.1. Transformations for two independent objects in the scene A, B.

The algorithm for the procedure described above, is shown in Algorithm 3.4.1.

Algorithm 3.4.1. Algorithm for the Inter–object Shading Effect.

For every receiving object A
    For every occluding object B
        Transform A's object space coordinates to B's texture space coordinates
        Call the shader for object A using the tranformed A coordinates, applying volume B to them
    End
    Accumulate results using additive blending
End

4. Tests

To verify the algorithms presented in section 3, some tests needed to be done, which shall be presented in this section. More specifically, we will test the algorithm's real-time speed for object with many triangles, present how the visual result is affected, depending on the radius and the voxel space resolution used. The visual results of the algorithm are also going to be compared to the Monte Carlo non-real-time method, and finally, we will show that the algorithm's visual result for multiple occluders is very nice for both near and distant occluders.

All the tests were run on a Core Duo Extreme/2.6GHz desktop PC with an nVidia8800GTS graphics board and 2GB of RAM.

Two pictures rendered with the spatial ambient occlusion method are shown in *Fig. 4.1* and *4.2*.

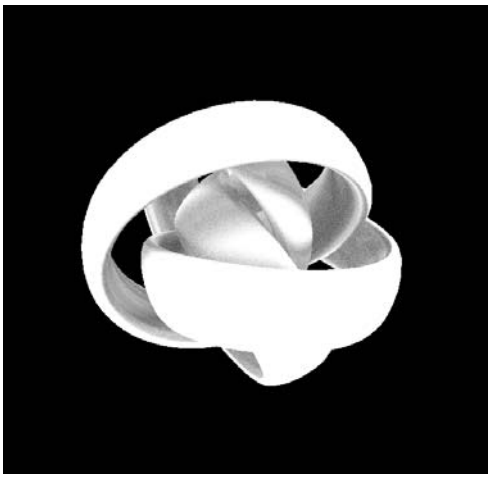


Fig.4.1. The knot (parametric primitive modeled with 3D Studio MAX) rendered using the spatial ambient occlusion method.

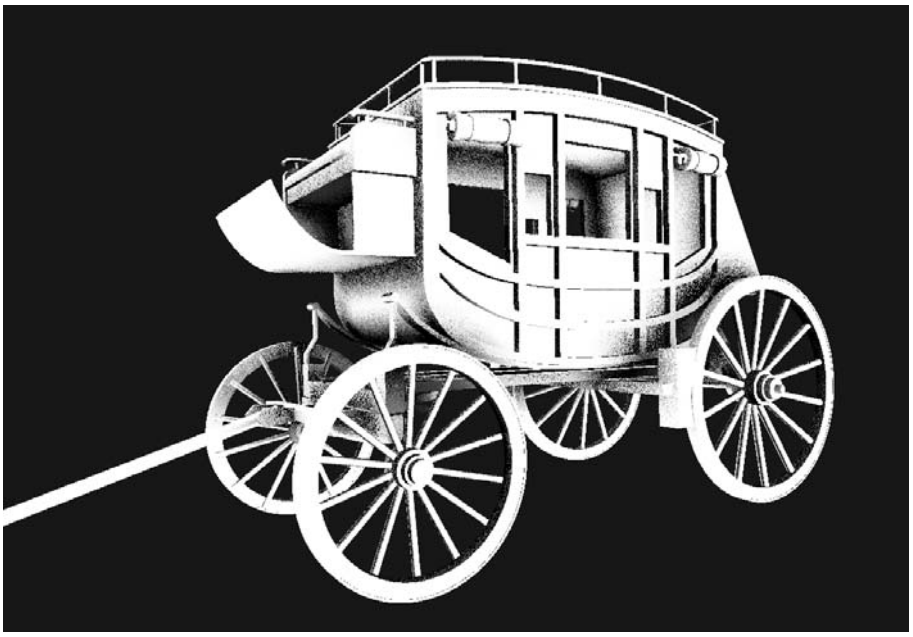


Fig.4.1. A coach model rendered using the spatial ambient occlusion method.

4.1. The Effect of Voxel Space Resolution

The voxel space resolution has some effect on the visual result. If the voxel space resolution is very low the object's illumination will be wrong, due to the fact that the spatial ambient occlusion is used, but as the resolution increases, the result tends to be the desired one. This happens because if the voxel space resolution is too low, then the number of voxels in the object's bounding box will be very small, and each voxel will be quite large. So all the proximity (positional ambient occlusion) values calculated for the voxels are very large, because most of the voxels are occupied by the object and those that are not will be very close to the object. Consequently, when the sampling takes place, no matter how far from the object (within a specified radius) the samples are taken, they will correspond to a voxel that is close to the object, having a large positional ambient occlusion value. If we have a folding object that is voxelized in low voxel space resolution, then all neighboring voxels will be occupied, thus not allowing the accumulative visibility value to drop between them and resulting to a very dark ambient occlusion illumination, which is wrong.

As the voxel space resolution increases, the number of voxels increases, so taking samples in a greater distance becomes meaningful.

This is obvious in *Fig. 4.1.1*, where for a voxel space resolution of 8^3, the result is very dark, but as the voxel space resolution increases, the visual result approximates the desired one.

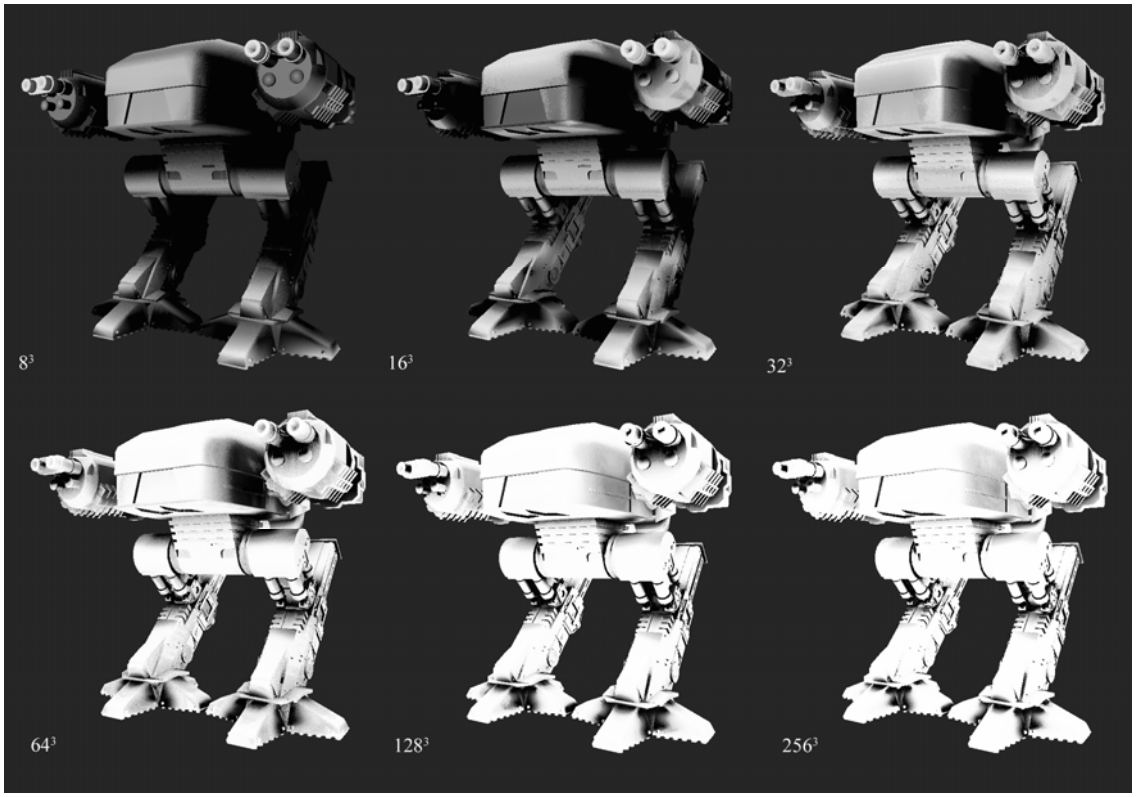


Fig. 4.1.1. The robot rendered with the spatial ambient occlusion method for different voxel space resolutions. It is obvious that the higher the resolution is, the more realistic the result will be.

As we have mentioned before (in Section 3.2.1), in our case the size of an object's bounding box determines the dimensions of the associated voxel space that will be used to store the proximity values. More specifically, the voxel space size is proportional to the ratio of the bounding box size and the reference bounding box, which, depending on the approach for the determination of the propagation radius (r_P), is either the bounding box of the union of objects or the maximal bounding box. This approach has been used for all the following tests, as far as the voxel space's resolution is concerned.

4.2. The Effect of Sampling Radius

As we have mentioned before (Section 3.3), the ambient occlusion effect range *R* represents the distance in which the proximity information calculated is taken into account in order to reconstruct the occlusion value. So we expect for R to have some effect on the visual result.

More specifically, when *R* is relatively small, samples are taken within a very small distance from every point on the object's surface, not allowing distant surfaces to shadow the desired surface; on the other hand, if the radius is relatively large, distant surfaces interact.

The above, is obvious in *Fig. 4.2.1*, were the inner part of a building is presented. We can see that when *R* is relatively small, then only the corners of the building are shaded. As the radius increases, a surface can be shadowed by more distant surfaces. In the fourth picture (*R*=1.7) we can see that the right wall of the building atrium shadows its left wall, creating in a visually pleasant and convincing result.

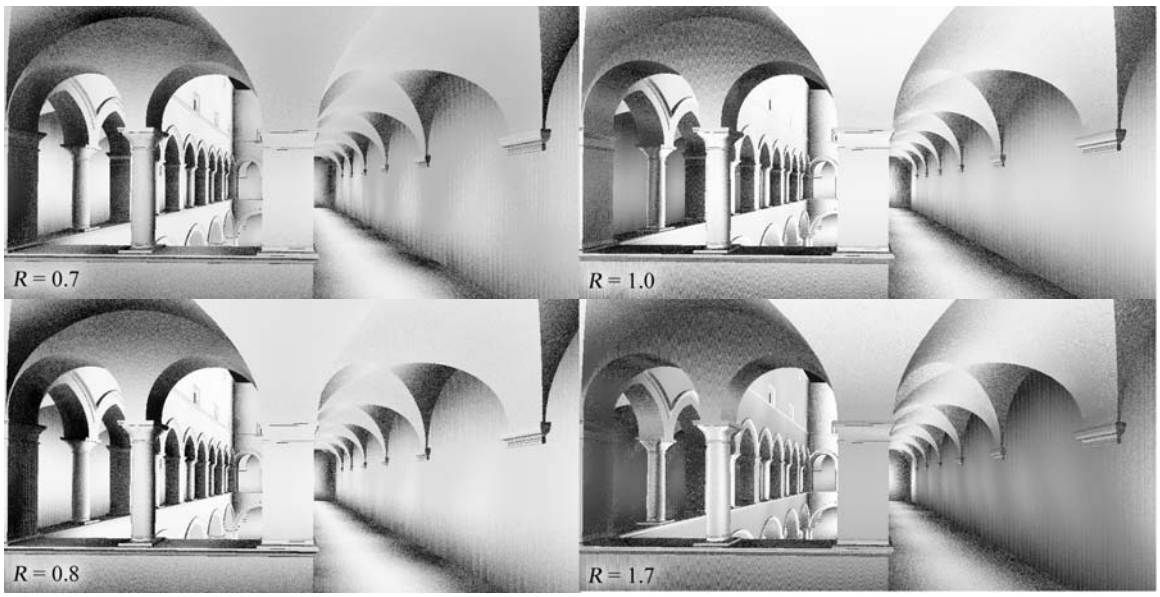


Fig.4.2.1. The effect of radius. The inner part of a building (the Sponza model) rendered using the spatial ambient occlusion method for real-time rendering. As the ambient occlusion effect range *R* increases, more distant occluders cast shadows on a receiving surface.

4.3. Speed

In this subsection, we will show that the algorithm presented is a very fast algorithm that can be used for real time applications having very large meshes.

In *Fig.4.3.1*, we can see an object consisting of 871,414 triangles, whose voxel space resolution was 128x128x64, rendered in a window, whose size was 1024x1024 pixels. The result is pleasing, and the speed was 135ms/frame, which is a speed that can be used for real-time rendering. In *Fig.4.3.2*, we can see an object consisting of 1,087,716 triangles, whose voxel space resolution was 64x128x64, rendered in the same resolution. The result, once again, is accurate and the speed was 133ms/frame.

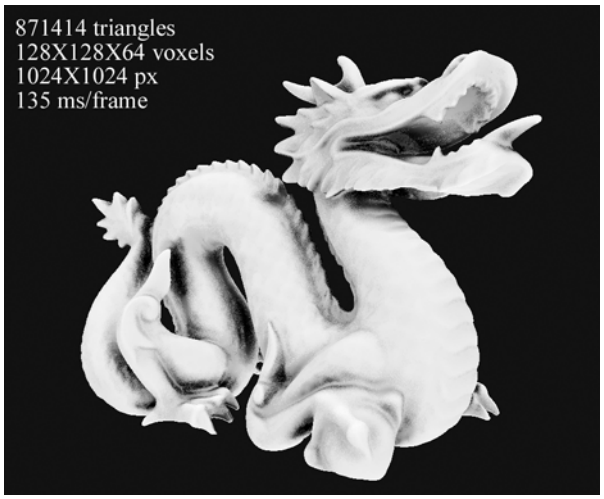


Fig. 4.3.1. The dragon model, consisting of 871414 triangles, rendered at 135ms/frame with the spatial ambient occlusion method in a window of size 1024x1024 pixels and a voxel space resolution of 128x128x64.

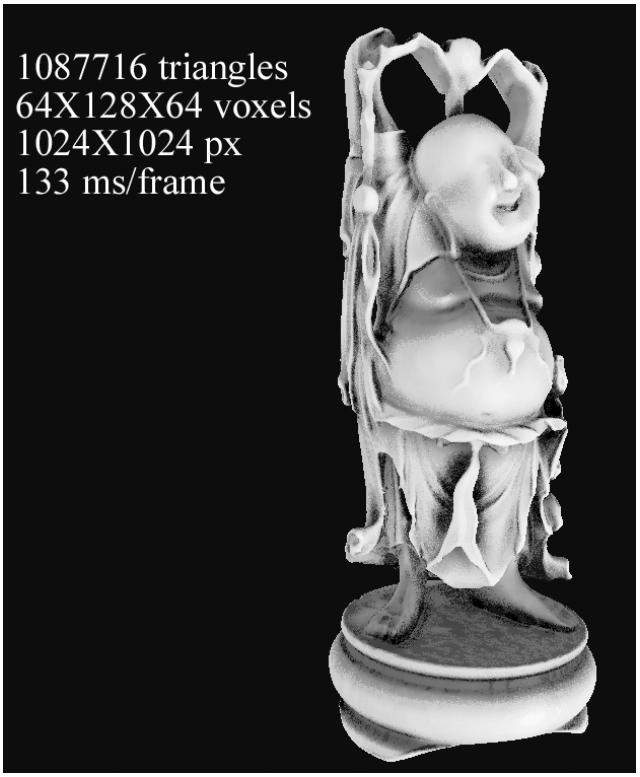


Fig. 4.3.2. The happy Buddha model, consisting of 1087716 triangles, rendered at 133ms/frame with the spatial ambient occlusion method in a window of size 1024x1024 pixels and a voxel space resolution of 64x128x64.

So the algorithm is independent of the number of triangles rendered[1], but it does depend on the number of objects rendered simultaneously in the same scene. More specifically, as shown in *Fig. 4.3.3*, as the number of object increases the time needed for every frame increases too.

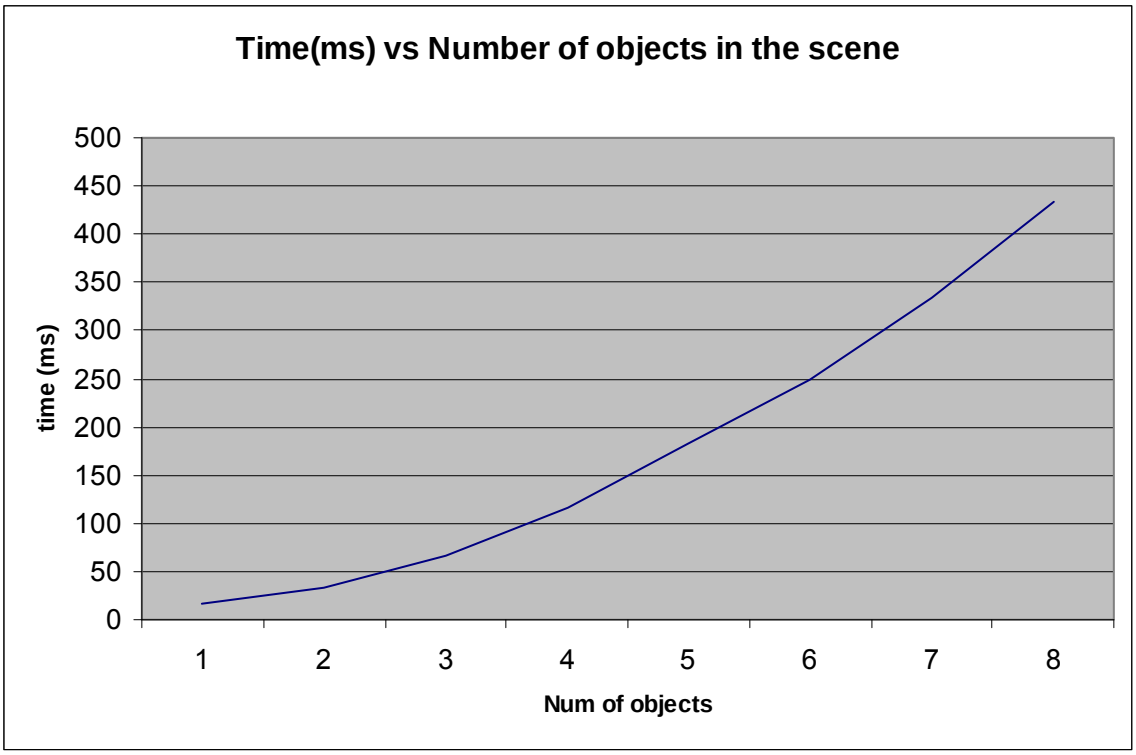


Fig. 4.3.3. Time needed for every frame in ms versus the number of objects in the scene. As the number of object increases the time needed for every frame increases as well.

We should also mention that the algorithm presented is not only a fast real-time algorithm; its pre-processing stage is also fast. For a constant number of triangles the pre-processing time depends on the voxel space's resolution (the higher the resolution, the more time is needed). This dependence is illustrated in Fig. 4.3.4, where we can see that as the number of voxels increases the time needed for pre-processing increases as well. The tests were made for a specific model (the robot, consisting of 19760 triangles), with standard propagation radius equal to the 10% of the maximal bounding box. Note that pre-computation time is split into voxelization time and proximity propagation time. The first depends on the number and size of primitives in the same manner as any rasterization method does, while the second is constant for a given voxel space resolution.

[1] The rendering speed is proportional to the fragment throughput of the graphics card, as in every constant time fragment shader.

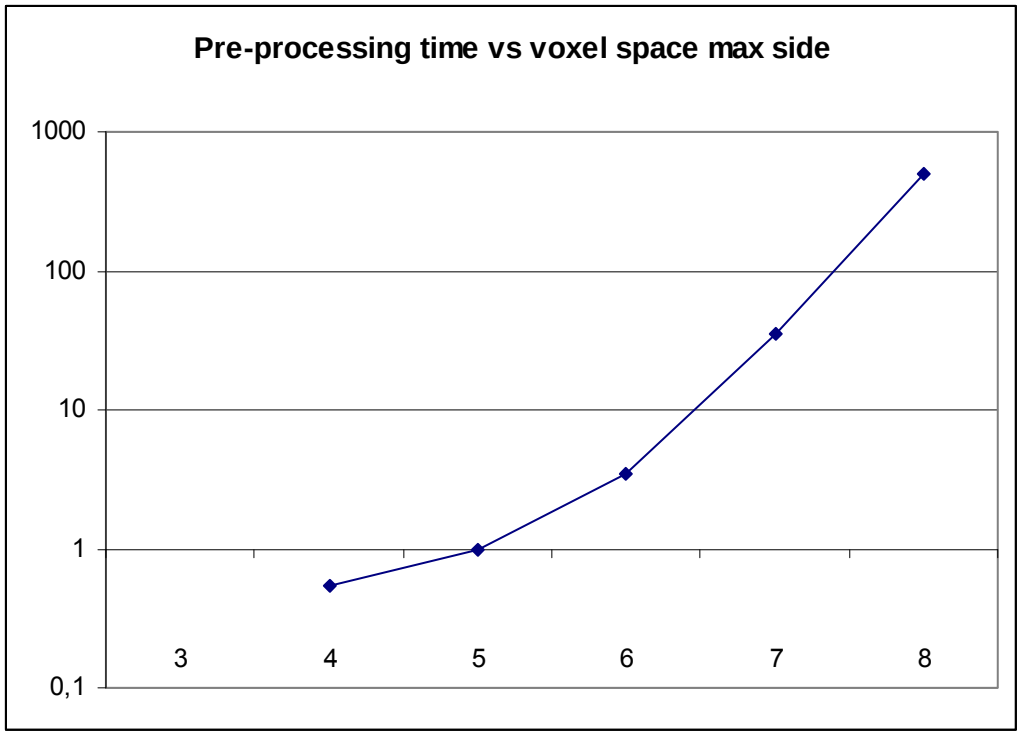


Fig. 4.3.4. Time needed for pre-processing in seconds versus the maximum side resolution of the voxel space. As the number of voxels increases the time needed for pre-processing increases as well.

4.4. Inter-Object Occlusion

In this subsection, we will discuss the effect of the algorithm's inter-object occlusion handling, to the final visual result.

In *Fig.4.4.1*, we can see four views of a hangar consisting of 7 individual objects, all produced by the algorithm presented. The first one shows an outside view, while the other 3 show details from its interior. All objects inside the hangar are different meshes and are handled as independent objects, shading each other according to the algorithm 3.4.1 for inter-object shading effect.

In these four pictures, we can see that the algorithm behaves well, for both, nearby and distant occluders, creating a detailed high-frequency and a distant low-frequency ambient occlusion effect respectively, thus, not requiring two different approaches, as is the case with the ND-buffer method [Shan05].

Since both the inside and the outside of the building are shaded correctly, we can assume that the algorithm can be used to illuminate arbitrary convex or concave surfaces.

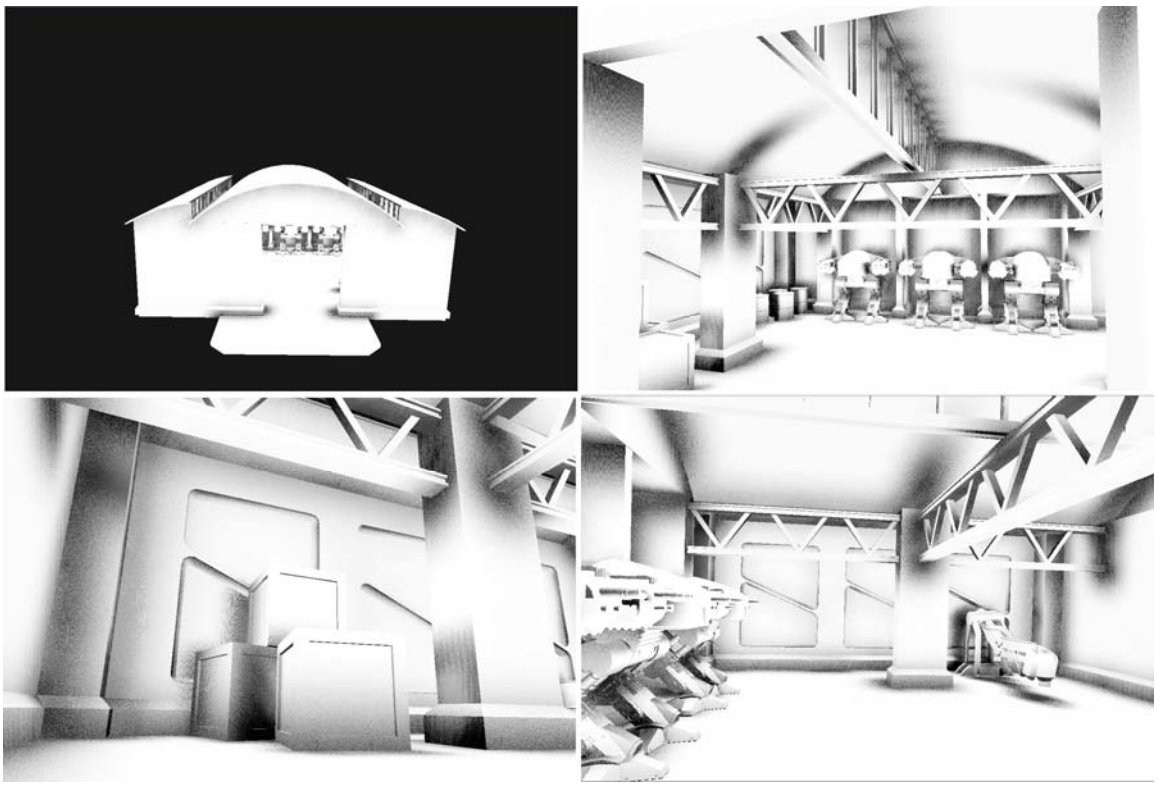


Fig.4.4.1. The inter-object occlusion effect. Four views of a hangar, all produced by the spatial ambient occlusion method presented.

4.5. Monte Carlo Ray Casting vs Spatial Ambient Occlusion

The spatial ambient occlusion algorithm presented, gives nice visual results, even though it is an approximation of the typical ambient occlusion calculation (Monte Carlo ray casting method).

In *Fig. 4.5.1, 4.5.2* and *4.5.3* a comparison is made between the two methods. Obviously the Monte Carlo ray casting method is more realistic, since the one of spatial ambient occlusion has relatively hard shadows, but the spatial ambient occlusion is a rather good approximation. After all, it should be noted, that although the Monte Carlo method gives excellent visual results, it is a non-real-time rendering technique, while the spatial ambient occlusion method's real-time speed is 67 ms/frame, which can be used for real-time applications.

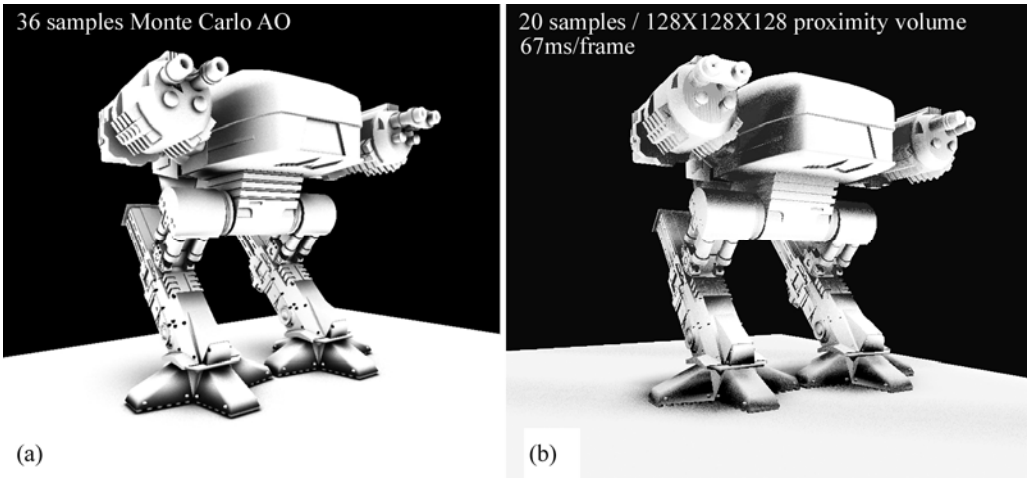


Fig.4.5.1. Monte Carlo ray tracing vs Spatial Ambient Occlusion for a robot (single object). (a) Monte Carlo method has been used, for non-real-time rendering, with 36 samples, (b) Spatial Ambient Occlusion method has been used with 20 samples on a 128x128x128 proximity volume resolution, for real-time rendering (67ms/frame)

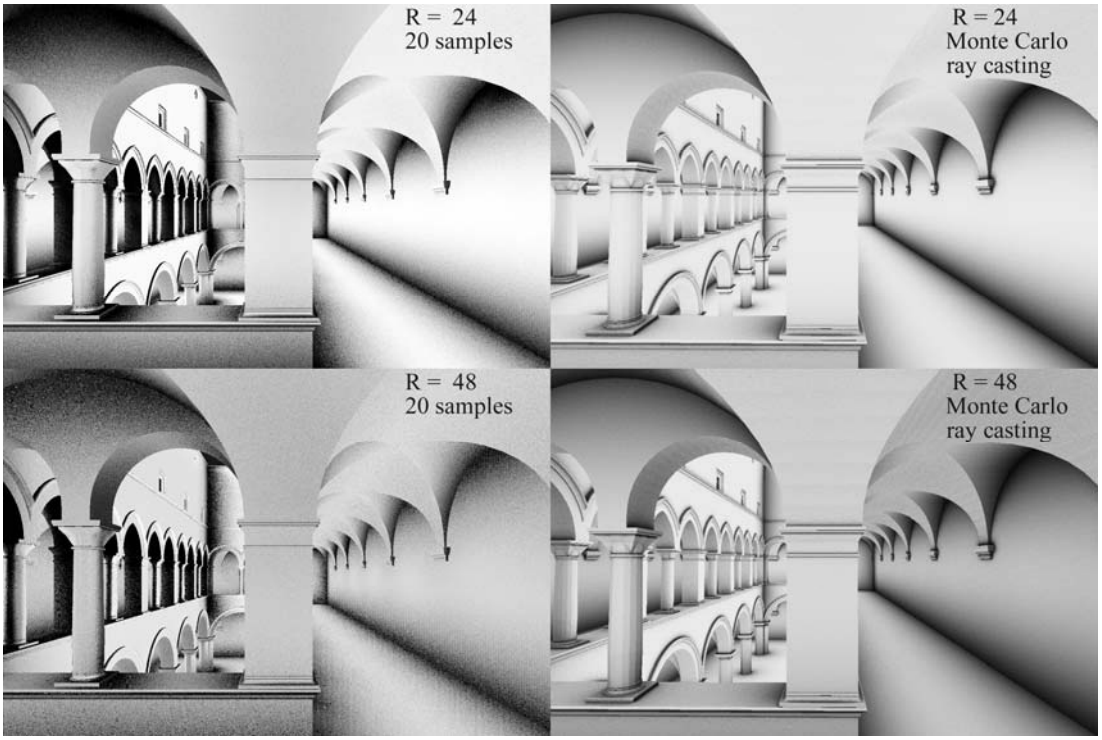


Fig.4.5.2. Monte Carlo ray casting vs Spatial Ambient Occlusion for the Sponza model. First row: The same scene rendered first with Spatial Ambient Occlusion method for real-time rendering with 20 samples, and then with the Monte Carlo mehod for non-real-time rendering, both using R=24. Second row: The same scene rendered first with Spatial Ambient Occlusion method for real-time rendering with 20 samples, and then with the Monte Carlo mehod for non-real-time rendering, both using R=48.

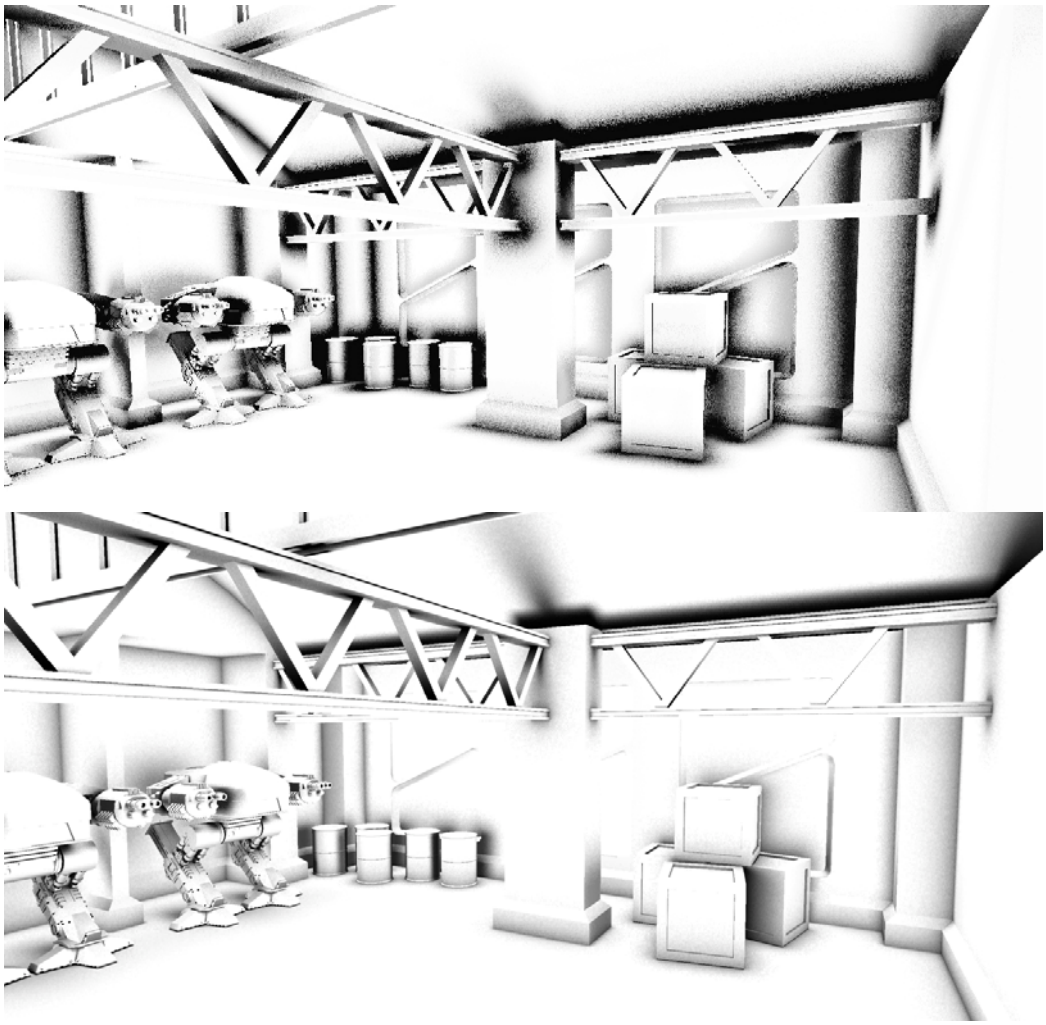


Fig.4.5.3. Monte Carlo ray casting vs Spatial Ambient Occlusion for the hangar model. The same scene rendered first with Spatial Ambient Occlusion method for real-time rendering (upper picture) and then with the Monte Carlo ray casting method for non-real-time rendering (lower picture).

5. Conclusions and Future work

We have presented an algorithm for pre-calculated visibility, that can be used to produce a visually pleasant ambient occlusion and lighting approximation. The main advantage of the proposed algorithm is that it does not use directional data (and consequently it doesn't have to store such data), and most of the computations are done in the pre-processing stage; so this is a fast algorithm for real-time graphics, that uses a comparatively small amount of memory storage. Furthermore, it does not suffer from view-dependency, nor has it the local effect depth-map-based ambient occlusion algorithms have.

The algorithm also has good scalability, as the volume distance data that are generated in a pre-processing stage only depend on the new objects. The proposed algorithm also, works well for both complex and simple geometry and it can be used to illuminate arbitrary convex or concave surfaces and it poses no limitation on surface connectivity.

As we have shown in section 4, the algorithm presented is independent of the number of triangles in the scene (as far as the proximity calculation is concerned). With this method, ambient occlusion in the interior of objects (e.g. inside a room), self occlusion, and inter-object occlusion can be calculated, to produce a visually pleasant final result.

An interesting consequence of the proximity value storage used in our algorithm, not implemented yet, is that it is possible to incorporate in the same ambient illumination calculation the contribution of direct diffuse illumination, such as light emitters of various sizes, shapes and light emission distributions, offering an integrated shading approach. So another advantage of the method presented is that it can handle active and passive illumination effects uniformly.

And although it is a very fast method, which can be used for real-time applications, it can be even faster; Fragment culling can be added to impressively increase the speed of the real time part of the algorithm.

The main defect of this algorithm, is that it cannot be used to calculate the ambient occlusion of deformable objects, as many other real-time algorithms. It also suffers from the presence of noise as it performs stochastic sampling.

6. References

Ambient Occlusion:

[Bunn05] Michael Bunnell. "Dynamic Ambient Occlusion and Indirect Lighting." In *GPU Gems 2*, edited by Matt Pharr, pp. 223–233. Addison Wesley, 2005.

[Gait08] A. Gaitatzes, Y. Chrisanthou, G. Papaioannou. "Presampled Visibiliy for Ambient Occlusion". Proc. WSCG 2008, Journal of WSCG, 16(1-3), pp.17-24, 2008.

[Kirk07] A.G.Kirk, O.Arikan. "Real-Time Ambient Occlusion for Dynamic Character Skins". ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. Seattle, WA, April 30 - May 2, 2007.

[Kont05] Janne Kontkanen and Samuli Laine. "Ambient Occlusion Fields." In *Symposium on Interactive 3D Graphics and Games*, pp. 41–48, 2005.

[Malm06] M. Malmer, F. Malmer, U. Assarsson, N. Holzschuch, T. Cog, Projets Artis. "Fast precomputed ambient occlusion for proximity shadows". Journal of Graphics Tools. 2006.

[Shan05] P. Shanmugan, O.Arikan. "Hardware Ambient Occlusion Techniques on GPU". Symposium on Interactive 3D graphics and games. 2005.

[Zhuk98] 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.

Voxelization:

[Chen98] H. Chen and S. Fang. Fast voxelization of 3D synthetic objects. *Journal of Graphic Tools*, 3(4):33–45, 1998.

[Kara99] E. Karabassi, G. Papaioannou, and T. Theoharis. A fast depth-buffer-based voxelization algorithm. *ACM Journal of Graphics Tools*, 4(4):5–10, 1999.