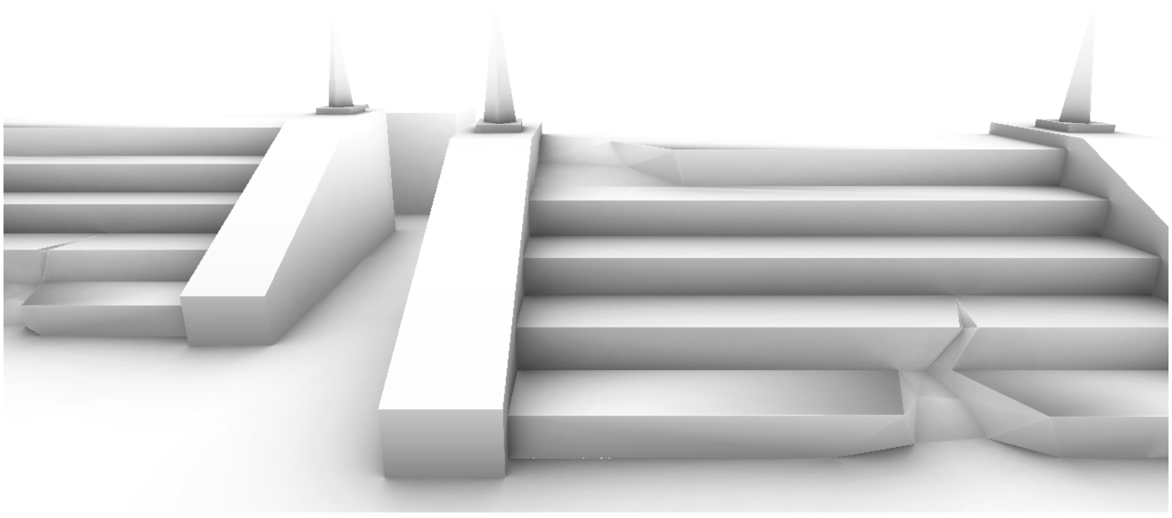

Realistic Ambient Occlusion In Real-Time

Survey of the AOV algorithm and propositions for improvements

Master of Science Thesis in Computer Science

JOAKIM CARLSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, 2010

Realistic Ambient Occlusion In Real-Time
Survey of the AOV algorithm and propositions for improvements

Joakim Carlsson

© Joakim Carlsson, 2010.

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
Image illustrating some of the effects achieved with the technique presented in this thesis.

Department of Computer Science and Engineering
Göteborg, Sweden 2010

Abstract

Ambient Occlusion is an area that has seen increased activity in the last few years and is an important aspect of reality in 3D-graphics. Rough approximation algorithms that are fast enough to be used in real games have appeared, but realistic solutions still have a way to go.

This thesis covers an evaluation of the recently published Ambient Occlusion Volumes algorithm and presents various attempts of similar approaches to increase the algorithm performance. While no major breakthrough were made, it presents some new ideas and further examines some of those presented in the original paper. The algorithm is currently not fit for real-time rendered ambient occlusion except for applications dedicated to that purpose, but this might change in the near future, as it relies heavily upon the GPU which is a fast advancing field of technology.

Sammanfattning

Kontaktskuggor är ett område som har sett ett ökat intresse de senaste åren. Grova approximeringsalgoritmer har dykt upp, men verklighetstrogna lösningar är inte riktigt där än.

Denna rapport presenterar en undersökning av den nyligen presenterade algoritmen Ambient Occlusion Volumes. Den går även igenom ett flertal försök med liknande algoritmer i syftet att öka prestanda vid körning. Inga större genombrott har gjorts, men den presenterar ett antal nya idéer och går även in djupare på vissa områden än originalrapporten. Algoritmen är ännu för långsam för att användas till att rendera kontaktskuggor i realtid annat än i program som inte behöver övrig tillgång till grafikkortet. Detta kan emellertid komma att ändras inom en snar framtid då grafikkortsteknologin går starkt framåt.

Contents

1 Introduction

Ambient illumination is a term that describes the indirect illumination of distant light, such as light from the sky, or light reflected on distant objects. Ambient occlusion refers to how nearby objects can occlude parts of this ambient light, producing indirect shadows or contact shadows. In effect, this is the darkening of curved or closed surfaces in a scene, for example if we have a car on a cloudy day, we will have a darkening underneath it due to the ambient occlusion (the car blocks the ambient light that would otherwise have reached the ground).

Computing this term efficiently is currently one of the most important topics in real time graphics, as it greatly improves the visual quality of images, not only making it more physically correct but also providing a visually pleasing experience.

Much of the current research in ambient occlusion deals with utilizing the GPU to produce ambient occlusion in images. An early production-ready solution was the CryTek Screen Space Ambient Occlusion (SSAO) presented in [Mittring, 2007] and utilized in their graphics engine *CryEngine 2*. This was followed by a great number of papers on improvements and variations on this technique, such as [Bavoil and Sainz, 2009] and [Bavoil et al., 2008].

A slightly different approach to this, but still in screen space, was introduced in [McGuire, 2009], called Ambient Occlusion Volumes. The algorithm is analytical and produces smooth and near ground truth results at impressing frame rates, but can not be considered real-time for production use. This thesis examines the algorithm and introduces several attempts to improve it, with the goal of making it suitable for real-time applications, such as games, on modern hardware.

1.1 Previous work

A technique described in GPU Gems 2 [Matt Pharr, 2005] treats polygon meshes as a set of surface elements. The benefit of this is that calculations are not needed between each and every element. A fast approximation is then done to compute the shadowing from blocking geometry.

A further improvement to the technique described in GPU Gems 2 is suggested in GPU Gems 3 [Boubekeur and Schlick, 2007]. They give a demonstration on a few changes to the algorithm that increases the usefulness and the robustness of the technique.

Another paper which makes use of ideas presented in GPU Gems 2 [Matt Pharr, 2005] is paper named "Accelerated Ambient Occlusion Using Spatial Subdivision Structures" [Wassenius, 2005]. The technique adds geometry to a spatial subdivision structure (an octree) and traverses it to find nearby occluders.

Doing ambient occlusion in screen-space has recently become very popular. On of the first introduced was the Screen-Space Ambient Occlusion (SSAO) [Mittring, 2007], in which the idea is to approximately calculate the ambient occlusion of a pixel by sampling the a depth buffer. A paper called Screen-Space Directional Occlusion (SSDO) [Ritschel et al., 2009] builds on this technique, and enhances it with directional information. The directional information can then used to calculate local indirect illumination, or together with an environment map; directional shadows. This provides an even more accurate solution than the SSAO solution, with little computational cost.

By precalculating a set of factors around an object, from which the ambient occlusion factor can be derived, [Kontkanen and Laine, 2005] were able to perform ambient occlusion between objects very quickly. [Malmer et al., 2007] built upon this idea, but stored the occlusion factors in a 3D grid instead, which made the algorithm even faster and can handle self-occlusion in the same pass.

[Evans, 2006] also uses 3D textures to estimate ambient occlusion, but instead of per object they use one large grid for the whole scene, which they can quickly create and then use to estimate ambient occlusion.

[Reinbothe et al., 2009] introduces the idea of voxelizing the scene each frame, and then using this information to calculate the ambient occlusion, achieving interactive frame rates.

1.2 Problem statement

This section will give a brief analytical motivation for ambient occlusion. The rendering equation is commonly defined as:

$$L_o(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + \int_\Omega f_r(\mathbf{x}, \omega', \omega) L_i(\mathbf{x}, \omega')(\omega' \cdot \mathbf{n}) d\omega'$$

where L_o is the light exiting from a surface point $\mathbf{x}$ in the direction ω, L_e is the emitted illumination, f_r is the material properties, L_i is incoming light and $\mathbf{n}$ is the surface normal.

The incoming light can be divided into two terms: "near" and "ambient" light, depending on how far the light travels before it hits the surface. This means the space is divided into two parts, the near space which is a sphere with a radius d, and the ambient space which is everything outside it.

$$L_o(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + L_n + \int_\Omega f_r(\mathbf{x},\omega',\omega)V(\mathbf{x},\omega)L_a(\mathbf{x},\omega')(\omega' \cdot \mathbf{n})d\omega'$$

where $V(\mathbf{x},\omega)$ is the visibility function, which is 1 if a ray from $\mathbf{x}$ in the direction ω is unobstructed for at least a distance d, and otherwise 0. The L_n term describes the "near" incoming light, such as direct light and indirect light (bouncing from nearby surfaces), and is described in more detail in [McGuire, 2009]. The last term describes the ambient illumination of the surface point, and a common approximation is:

$$\pi\left(\int_\Omega f_r(\mathbf{x},\omega',\omega)L_a(\mathbf{x},\omega')d\omega'\right)\left(V(\mathbf{x},\omega)(\omega'\cdot\mathbf{n}))d\omega'\right)$$

The first factor of this can be precomputed with only a small error for diffuse surfaces, thus only the second factor needs to be computed. The second factor could be described as the accessibility of the surface point, i.e. a number from 0 to 1 describing how much ambient light reaches it. Ambient occlusion is generally defined as one minus the accessibility. Calculating this integral efficiently, with an error as small as possible, for each surface point visible on the screen is the primary problem that ambient occlusion algorithms try to solve.

2 Ambient Occlusion Volumes

The AOV (ambient occlusion volumes) algorithm calculates the ambient occlusion term in the rendering equation. It operates in screen space and makes use of a deferred rendering pipeline. We will now explain this method briefly. For a full explanation, this method is thoroughly explained in [McGuire, 2009].

2.1 Deferred rendering

A deferred rendering pipeline is used to create intermediate geometry buffers which is of high importance to the AOV algorithm. A common need, is to find the world position of a pixel in screen space, which is where the AOV algorithm operates. In the geometry buffer, the worldspace normal is stored which is also needed for the AOV algorithm.

2.2 Ambient occlusion volumes

Every polygon in the scene is considered to be a potential occluder to a pixel. To limit the amount of pixels to be evaluated, a volume is extruded from every polygon which makes up for the space in which pixels are considered. The volumes are being extruded both horizontally and vertically from the polygon. This is done in the geometry shader. For each vertex in the polygon, an extension vector is being calculated. The normal, which is computed in the geometry shader, and the extension vectors are then used to extrude the polygon vertically and horizontally respectively. It is important to store the magnitude to which we are extruding the volumes in order to later scale the amount of occlusion added to a pixel according to the distance from the polygon to the pixel in world space in order to avoid hard edges on the volumes.

2.3 Algorithm

For each pixel found by rasterizing the volumes, the corresponding pixel in worldspace is being calculated using the depth value stored in the geometry buffer. The polygon, which in this case is a triangle, belonging to the rasterized pixel is then considered to be a potential occluder to the worldspace pixel. There are four cases which can arise from this setting.

1. The triangle is below the surface to the normal of the worldspace pixel
2. Two vertices in the triangle is above the surface to the normal of the worldspace pixel
3. One vertex in the triangle is above the surface to the normal of the worldspace pixel
4. All vertices are above the surface to the normal of the worldspace pixel

In the first case, no possible occlusion could be dealt to the pixel. In the second and third case, clipping has to be done between the triangle and the surface to the normal of the worldspace pixel in order to find the part of the triangle which is above the surface. In the forth case, the whole polygon is considered.

In the final step of the algorithm, an occlusion value is calculated for each considered pixel. Subtractive blending is used in order to combine occlusion values. For the considered world space pixel, a hemisphere, based on the world space normal is visualized to explain the following projections. The triangle is projected to the surface of the hemisphere, and further projected down to the surface of the normal to the worldspace pixel. The area of the projected triangle directly corresponds to the amount of occlusion which is weighted with a value in order to avoid hard edges.

3 Suggested improvements

3.1 Pre-calculated self-occlusion

If we look at the ambient term of an object, we can split it into two components: self-occlusion and occlusion from other objects. For static geometry, the self-occlusion term will never change, since it only depends on the geometry itself. This means that we should be able to pre-calculate this term and only calculate the occlusion from other objects live. Dynamic geometry (such as, for example, animated skinned-mesh objects) are handled as normally done by the AOV algorithm.

Two major parts of this optimization can be identified; calculating the self-occlusion efficiently and utilizing this pre-calculated self-occlusion in the program. The different stages of the algorithm is visualized in Figure 1.

This technique is similar to light maps [Beam, 2003], with the difference that light maps often refer to whole-scene pre-calculated lighting, and not only self-occlusion. Light maps may also handle direct illumination.

3.1.1 Algorithm outline

1. (Offline) Create **occlusion maps** for every object type
2. (Each frame) Create an **object id buffer**
3. (Each frame) Render the scene using normal lighting, and use the occlusion maps to provide self occlusion
4. (Each frame) Render the Ambient Occlusion Volumes on top of this, but skip pixels which would calculate self-occlusion

3.1.2 Efficiently pre-calculating the self-occlusion

The first part of the optimization is the pre-calculation of the self occlusion. What we get from this is a texture which we call the occlusion map. This texture contains values defining how much light, or occlusion, points on the surface of the object receives. It is important to UV-map the object in such a way that no faces overlap in the UV-map, otherwise those texels occlusion value would represent several faces occlusion value.

The occlusion map can be calculated in a lot of ways, and it is unimportant to the second part of the algorithm exactly how they are constructed. We have devised one algorithm that calculates the occlusion map on the GPU using the AOV algorithm, which has the benefit of producing results that are very similar to calculating the AOV self-occlusion term live, meaning that the difference between an image with pre-calculated self-occlusion and an image with normal AOV is very small (as can be seen in Figure 1).

The algorithm can be outlined as follows:

1. Calculate a normal and a world position buffer in texture space (using the UV-coordinate of the vertex as position, and outputting the normal and world position as texcoords)
2. Run the AOV algorithm on an empty buffer, but instead of outputting volumes output quads covering the whole buffer.

The reason this works is because of how the AOV algorithm works. The AOV pixel shader has the following input: world position and normal of pixel we are trying to shade and the triangle that we want to calculate how much it occludes that pixel. The world position and normal are simply sampled from the calculated buffers in stage one, and the triangle is simply output in the geometry shader (as normally done) in the second stage.

A few points should be noted though:

- You can choose to use or omit the g_p function. Omitting it is mathematically equal to sampling towards infinity when calculating the AO term. This might however not always be desirable, for example inside a room model, since the room would then be rendered completely black (the AO_p term will be one).
- The P triangles outputted in the geometry shader can be offset by a small value along the m_k vector to prevent "edges" of under-occlusion in the occlusion map.
- After the algorithm is done, we can run a "padding" algorithm on the texture which basically fills texels that are outside the faces on the occlusion maps with nearby texel information, similar to the "push-pull" algorithm from [Grossman and Dally, 1998]. This prevents incorrect color "bleeding" when sampling the texture.

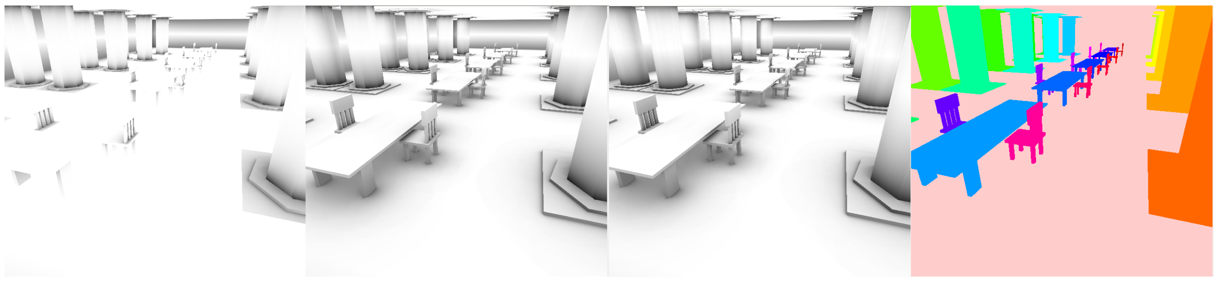
Pre-calculated self-occlusion
Pre-computed self-occlusion + occlusion from other objects
Standard AOV
Visualization of Object ID Buffer

Figure 1: *Pre-calculated self-occlusion visualizations*

3.1.3 Rendering

To utilize the pre-calculated self-occlusion in the live rendering, we have to perform a few additional steps to the original AOV algorithm. The general idea is to skip calculating self-occlusion on objects, and use the pre-calculated values in those places instead.

The first step is to calculate what we call an **object id buffer**. This buffer is a screen resolution buffer of integers, each integer representing the id of the object at that location. Next we clear the screen and then draw the scene normally, but with the self-occlusion maps applied to all objects. This will produce an image with only self occlusion, so after that we render the AOV algorithm to calculate occlusion from other objects, but with a modification; we only draw a pixel if the id of object we are working with is not the same as the id in the object id buffer.

3.1.4 Performance

The early measurements of this technique was made on a Radeon HD 3470, which is a graphics card with a fairly low fillrate (3.2 GPixels/s). All our test scenes were fillrate bound on this card.

We started out implementing the id buffer as an integer texture, which we rendered to in a separate pass, and then we simply read this texture in the AOV pixel shader and discarded pixels which had the same id as the object that was being handled. However, this proved very slow and we only saw a rough 10% performance gain compared to standard AOV.

We then implemented the id culling by writing id's to the stencil buffer while doing the normal rendering, and then using the stencil buffer to cull away the pixels with the same idea, which turned out to be much faster. In many of our test scenes we saw performance gains of around 50% (i.e. twice the framerate).

Measuring the overdraw we noticed a similar relationship; in many scenes the overdraw was halved compared to normal AOV, which explains the performance gain on this graphics card.

3.1.5 Conclusions

This technique is fairly straight forward to implement, and produces almost equal (if the occlusion map resolution is low), equal or better (the d variable can be arbitrarily large in the occlusion map, or g_p omitted) results than the normal AOV algorithm. The occlusion map generation is fast, and can be compiled offline once and then simply loaded from the hard drive.

The drawbacks are that it requires a UV-mapping with no overlapping faces (which means we either have to generate those maps in the modelling software, or compute them in some way; we created them in the modelling software for simplicity) and the extra memory usage of the occlusion maps. Another concern is that the stencil buffer can only store 8-bit values, which means that we can only handle 255 objects at a time, more than that and we would have to recalculate the stencil id buffer every 255th object for the next 255 objects.

3.2 Clipping

When computing contributions from polygon p to pixel x, one must find the parts of p that can possibly shade x. If all vertices of p are below the surface to the normal of x, p can not shade x. If all vertices lie above the surface to the normal of x, p fully shades x. However, clipping has to be performed on p when p intersects the surface to the normal of x. In our case, all polygons are triangles so there are two cases where clipping is needed. One where two vertices are above the surface to the normal of x and another where only one vertex is above the surface to the normal of x.

This is a simple task but since it is run for every pixel it has to be done fast. We found two analytical techniques. One suggest in the paper which is developed by GPU GEM, and another one developed by ourselves. We also tried to find an approximation algorithm for solving this problem in order to trade-off visual appeal for performance.

3.2.1 GPU-GEM

This technique is suggested for use in the latest version of ambient occlusion volumes. As the problem itself, the technique is straight forward. The algorithm goes through a series of if-statements in order to find the correct setting. To find the points of intersection, this technique computes the distances from the vertices of the triangles to the plane. This is illustrated in figure 2.

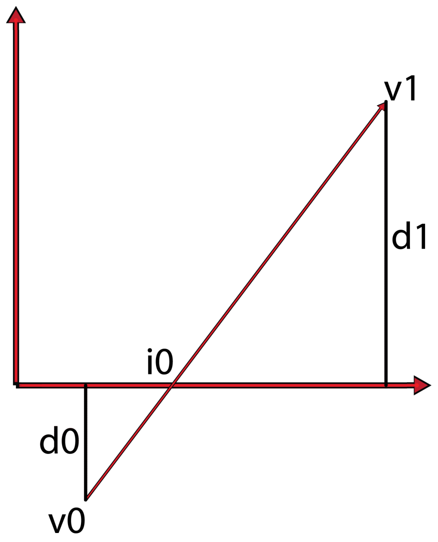

Figure 2: Distances d_0 and d_1 are calculated in order to find the intersection point i_0.

In order to find the intersection point we simply calculate it:

$$i_0 = (d_0/(d_0 - d_1)) * (v_0 - v_1)$$

They choose to structure their if-statements in a logical way that searches to find a setting of vertices which matches the actual case. The way of doing this is the key difference between our algorithm and theirs. Their structure looks as follows:

```
if(v0 is above)
{
    if(v1 is above)
    {
        if(v2 is below)
        {
            //compute intersections from
            //v2 to v1
            //v2 to v0
        }
        else if(v2 is above)
        {
            //all above
        }
    }
}
... etc ...
```

3.2.2 Our method

The main difference between this method and GPU GEM's method is how the if statements are structured. In our algorithm, we first count the number of vertices that are below the surface to the normal of x. By doing that we can split the if-statements into blocks, each representing a number of vertices below the surface to the normal of x. The structure looks as follows:

```
if(below == 3)
    //all below
if(below == 0)
    //all above
if(below == 2)
    //find which vertex is above
if(below == 1)
    //find which vertex is below
```

When there are two vertices below the surface we check which vertex is above to know which setting that needs to be handled. We then compute the intersection points on the surface to the normal of x

and the vectors from the vertex above to the vertex below. When there are only one vertex above the surface to the normal of x, we look for that particular vertex and compute the intersection points in the same manner as in the previous case.

3.2.3 Approximation methods

There are no approximation methods easily available online, which is natural since the nature of the problem itself is so simple. We developed our own to gain some knowledge into how much visual appeal one would have to trade-off to boost performance. In our best implementation, we managed to gain performance but the visual appeal was greatly impaired which ruins the soul purpose of AOV. The algorithm looks as follows:

```
Input: Polygon p, Pixelnormal N
Output: Clipped polygon p'

for each Vertex v in Polygon p
    if(v below plane)
        move v up to plane
return true if any v is above plane
    else false
```

What is actually being approximated is the intersection points of the triangle and the plane. The algorithm always returns three vertices in difference to the analytical version which returns a quad or a triangle dependant on the setting of vertices. This is a result from how the intersection points are being approximated. In our algorithm we rise vertices below the plane until they lie in the plane. This is illustrated in figure 3.

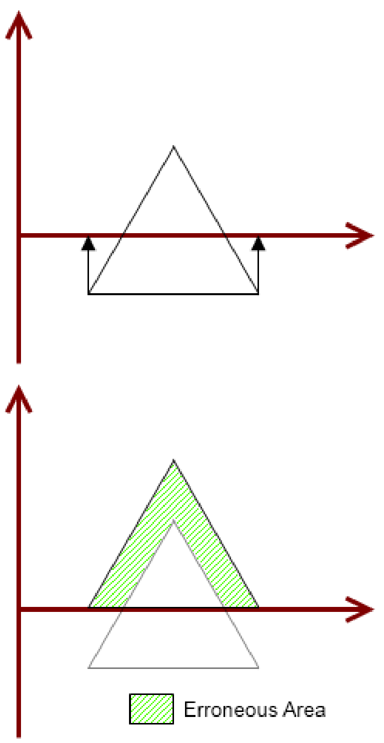

Figure 3: This figure illustrates how the clipping is approximated by translating the triangle until all points are above the surface of the normal to the pixel being shaded.

The error-area shown in figure 3, is dependant of the distances from the vertices to the plane. It can be reduced by doing additional computations at the expense of performance. Unfortunately the price is so high, the approximation algorithm no longer boosts performance in comparison to the previously mentioned methods. So the error-area is the final trade-off to gain additional performance.

3.2.4 Performance & Conclusion

The performance between our method and GPU-GEM's varies greatly between computers and can differ up to 20 %. We could not find any way to pre-determine which algorithm to use. It is very difficult to create an approximation algorithm that boosts performance while not killing off the visual looks. We were not able to develop a method that could out-perform GPU-GEM by a lot, but our most successful method seems to be at an advantage on most machines.

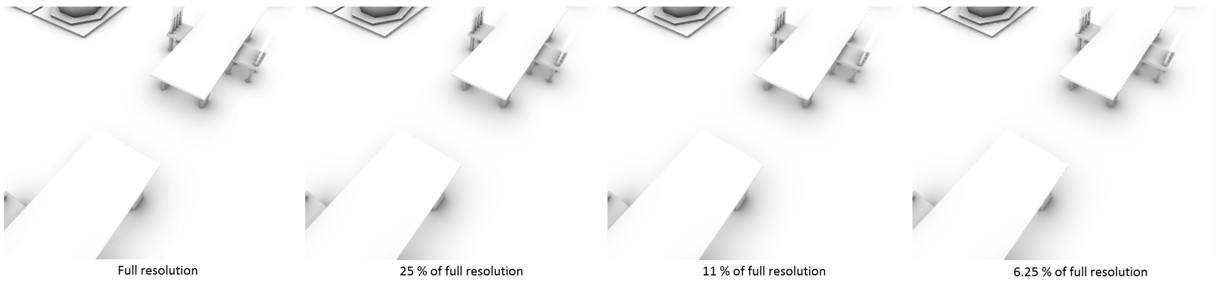

Figure 4: *A comparison between the ground truth and sampled pictures at various quality.*

3.3 Upsampling

A common way to improve performance for pixel bound methods is to reduce the size of the render target for that particular method. The output is written to an intermediate low resolution buffer and then rescaled using filtered sampling in order to reconstruct the information lost from the downsampling.

3.3.1 Bilateral filter

Many papers suggest ways to implement bilateral filters. What usually varies is the size of the kernels and the ways of weighing the normals, distances and depths.

In the AOV-paper a Gaussian function is used to weigh distances, normals and depths, and a $5x5$ kernel is used to sample from the low resolution texture.

A paper from 2008 [Lei Yang,] takes a similar approach to the implementation of the bilateral filter but with some interesting differences. To upsample the low resolution texture, they use a $2x2$ kernel. They also choose to weigh the distances differently by using a tent function rather than the typical Gaussian. Both papers use the Gaussian function to weigh normals and depths.

$$f(x) = ae^{\frac{-(x-b)^2}{2c^2}}$$

This is the Gaussian function that is used to weigh normals and depths. The width of the Gaussian curve is adjusted by modulating a. It needs to be carefully tuned in order to preserve sharp edges which is usually desired for visually pleasing images.

3.3.2 Implementation

To implement the suggested method in the 2008 paper for the AOV method, one has to create a high resolution and a low resolution geometry buffer. The low resolution geometry buffer should be of the size as the downsampled render target.

$$c_i^H = \frac{\sum c_j^L f(x_i,x_j) g(|n_i^H - n_j^L|,\theta_n) g(|z_i^H - z_j^L|,\theta_z)}{\sum f(x_i,x_j) g(|n_i^H - n_j^L|,\theta_n) g(|z_i^H - z_j^L|,\theta_z)}$$

This is the formula which is being applied to every pixel p_i. In our implementation a $2x2$ grid was used so this was run four times per pixel. x_i is the position of p_i in the downsampled texture and x_j is a sample in the grid. θ_z and θ_n is the weight of difference in the normals and depths respectively.

The 2008 paper mentions that small values for θ_z and θ_n are desired for maintaining sharp edges. In our implementation we found that $\theta_z = 0.1$ and $\theta_n = 0.1$ produced pleasing results.

3.3.3 Results & Performance

The results are showing huge performance gains from downsampling. This was to be expected since we are directly attacking the bottleneck of this method. When using smaller kernels we can further optimize the upsampling, in particular when using the $2x2$ instead of the $5x5$ grid. The trade-off in visual appearance is small, if even noticeable.

In order to get good results with a smaller kernel one has to be extra cautious about correctly tuning the weights to the Gaussian functions. In figure 4, visual appearance are shown when using different scales on the render target.

The downsampled scenes are rendered using a $2x2$ grid, weighing distances with a tent-function and depths and normals with a Gaussian function, as suggested in the paper by Yang. The performance differs a lot between different downsampling schemes. They are rendered in 290 ms, 105 ms, 65 ms and 45 ms respectively.

Using the same scheme, but with a $5x5$ grid the performance is highly affected and the quality gained is hardly noticable. The fourth picture rendered in 45 ms using the $2x2$ while it takes the $5x5$ grid 83 ms.

An obvious improvement is the second picture which is rendered three times as fast as the full resolution image while not losing any visual appeal. The third picture which downsamples the render target by almost 11% further increases performance by almost 100 %, while still not producing any noteworthy artifacts.

3.3.4 Conclusion

The $2x2$ grid suggested in Yang's paper show promising results in visual appeals and particularly in performance. The sampling in the $2x2$ is very fast in comparison with the sampling in the $5x5$.

3.4 Pre-calculated AO and clipping function

Looking at the pixel shader in the standard AO algorithm, we may identify a few things that are potentially worth pre-calculating. The input to the actual clipping and AO functions consists of three variables; the triangle, the surface normal at the point and the world space position of the point (equation 1).

t	:	triangle (3 vertices = 9 floating point values)
n	:	surface normal
x	:	world space position
ao	:	ambient term
h	:	hash value

Figure 5: *Definitions*

$$\begin{aligned} clip(t,n,x) &\to t' \\ AO(t',n,x) &\to ao \end{aligned} \tag{1}$$

Our idea was to use these values to calculate a hash value, which we would then use to look up the AO value (equation 2).

$$\begin{aligned} hash(t,n,x) &\to h \\ lookup(h) &\to ao \end{aligned} \tag{2}$$

Not only can we thus save the calculation of the AO_p value, but we can also save the very expensive clipping calculations, since this can be computed in the lookup table.

3.4.1 Naïve hash function

(t,n,x) consists of a total of 15 floating point values. The most simple solution would be to simply use these values to create a hash. If we map each of these values to only four bits, meaning they are limited to only 16 values, we will still have $16^15 = 1.1529215 * 10^18$ values, which is obviously much larger than any memory can hold.

3.4.2 Using spherical coordinates to map

A lot of the information in a naïve mapping would however be redundant. First of all we can get rid of x, and use $(t-x,n)$ instead, since it does not matter where in the world a triangle is, the calculations will be the same. Next, we may also transform the triangle with n, which gives us $(trans(t-x,n))$, a total of nine values. Further more, we may reduce these nice values by projecting the triangle onto the sphere and use spherical coordinates to describe the triangle (which only requires two values per vertex instead of three, since the radius is always one when the triangle is projected onto the sphere). Finally, the z-rotation of the triangle is unimportant so we may describe the values in reference to one of the vertices, leaving us with only five values $(a_\theta, b_\theta, b_\varphi - a_\varphi, c_\theta, c_\varphi - a_\varphi)$ (where (a,b,c) is the vertices of the triangle).

If we use one byte to store each ao value, and map each of these floating point values to four bits, then the map will be $1 * 16^5 = 1048576byte = 1MB$ large. Since 1D textures in directx only can be of 8192 texels in width, we use 3D textures, meaning we have to do additional mapping from these five floating point values to three integer values.

3.4.3 Performance evaluation and Conclusions

To do an early evaluation of the performance of this technique, we implemented a pixel shader that performed the lookup in an empty texture. The pixel shader is outlined in figure 6.

1. $t' = ToSphericalCoords(|t - x|)$
2. $n' = ToSphericalCoords(n)$
3. $t'' = Normalize(t' - n')$
4. $h = Map5FloatTo3Integer(t'')$
5. $ao = LookupInTexture(AOTexture, h)$

Figure 6: Pixel shader outline

Early measurements of this provided an performance increase of around 20-30%, which we consider low in relation to how complex the method is.

There are however alternative hashing functions which may prove more efficient; [Hr?dek et al., 2003] is one example.

3.5 Hard-limiting maximum number of overdraws

The biggest bottleneck of the AOV algorithm is the huge overdraw (around 10-20 times per pixel in average for many of our scenes). To counter this, a simple solution is to hard-limit the number of overdraws (by using for example the stencil buffer).

Implementing this is very straight forward, but, as can be seen in Figure 7, the artifacts are very large even at as large limits as 32 times overdraw (for example, the chairs are under-shaded at $n = 32$). Moreover, we could not measure any significant performance increase at all from this optimization, which we believe is due to the fact that the algorithm is heavily fillrate bound, meaning that even though it might be able to skip some pixel calculations the fillrate will still bottleneck the application.

3.6 SSAO using the AO_p function

As mentioned earlier, the AOV algorithm is heavily fillrate bound. The reason it is fillrate bound is because of the AO volumes used to identify pixels a triangle can potentially shade. These volumes overlap each other a great number of times, and since the results are blended together all of them have to be drawn, which produces a high overdraw count.

Our idea is to turn the relation around; instead of using volumes to identify pixels a triangle can shade, we try to find triangles that can shade a pixel for each pixel. This is inspired by the Screen Space Ambient Occlusion algorithms such as the Crytek SSAO [Mittring, 2007].

The algorithm can be outlined as follows:

1. Create a **triangle buffer**
2. Draw a fullscreen quad with the triangle buffer as input. For each pixel find triangles from the **triangle buffer** that can potentially shade it, and calculate the shading using the AO_p function.

3.6.1 Triangle buffer

The triangle buffer can be created by rendering the geometry using forward-rendering, and in the geometry shader appending information to each vertex as to which triangle it belongs to. We then output this information in the pixel shader, either in three targets (containing the three vertices of the triangle) or compressing each vertex into one float value.

This operation is very fast and lightweight, though it does scale with scene complexity.

3.6.2 Computing AO

We then proceed by drawing a fullscreen quad to compute the AO term of each pixel. For each pixel we sample a number of triangles from the triangle buffer. These triangles are then used in the normal AO algorithm, with the exception that we don't need to use the g_p function.

3.6.3 Advantages and Limitations

The primary advantage with this method is that it scales very well with scene complexity. The triangle buffer creation is the only part of the algorithm that is proportional to scene complexity.

The AO computation depends on three values; the screen resolution, the number of samples and the size of the kernel (since a larger kernel decreases cache performance).

However, the major drawback with this method is that it comes with a large constant factor; the pixel shader of the AO computation is rather heavy since we have to try to find the triangles that shade that pixel.

We also have the problem of choosing a kernel. Several kernels were tried out, amongst them a grid, random sampling and a grid with each sample offset with a random number, each of these scaled in size

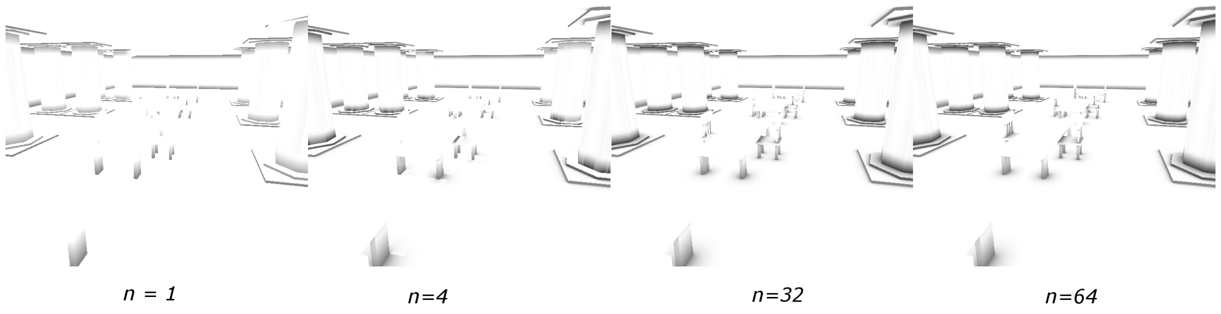

Figure 7: *Overdraw with hard-limited number of maximum overdraws (n).*

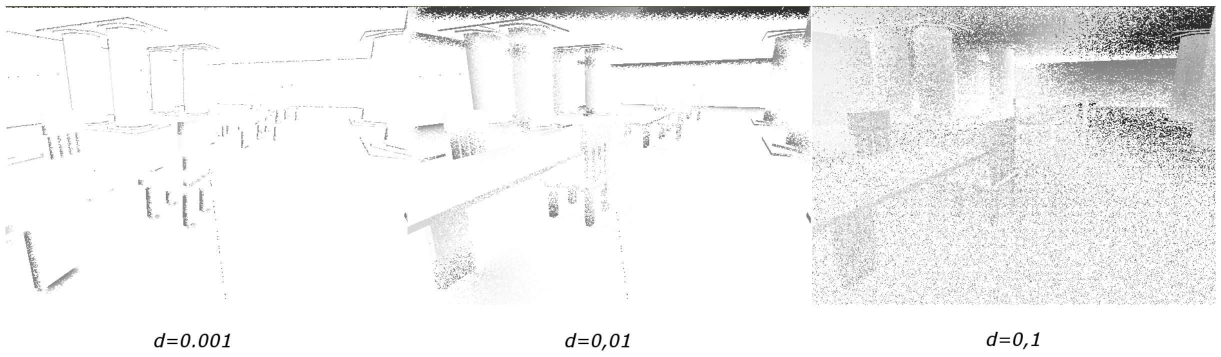

Figure 8: *SSAO using the AO_p function. Using a grid kernel with random offsets for each sample, scaled with the depth of the pixel and the constant d. Notice that no matter the value of d the kernel cannot find the triangles under the table (since we are sampling in screen space), and thus we will never have shading below the table.*

by the distance of the pixel. All of these had troubles finding appropriate sample spots (i.e. finding the potential triangles that could shade this pixel).

Above all, the problem is finding triangles which are not visible in screen space (as detailed in Figure 8). This could potentially be solved, at least partially by, for example, techniques described in [Bavoil and Sainz, 2009].

3.6.4 Conclusions

Although this technique has its advantages, the drawbacks makes it in its current state practically useless, as there exists various other SSAO techniques that would be preferred.

3.7 Ambient Occlusion lookup table using circle-approximation

Due to the high amount of overdraw for Ambient Occlusion Volumes, reducing the complexity of computations in the pixel shader is essential in order to reach good frame rates. This approximation technique utilizes pre-calculated values in order to remove the occlusion calculations from the pixel shader.

3.7.1 Main idea

The main idea behind this algorithm is to replace the hemisphere-projected triangles found in the pixel shader-stage of the original algorithm with circles of the same area. By doing this the work that needs to be done each frame is reduced to finding the circle approximation for each triangle processed in the pixel shader. The ambient occlusion can then be found by doing a lookup in a pre-calculated table.

The pre-calculations performed finds the Ambient Occlusion values for different values of the position angle θ and the circle size angle ψ and outputs these to a texture.

3.7.2 Circle approximation (Pixel shader)

Starting out with a triangle P in worldspace coordinates and a sphere around point $\vec{x}$, the triangle vertices are projected onto the sphere surface with new coordinates A, B and C. By doing this before finding the circle we do not have to take into account the angle of P in regard to the sphere. We find the circle radius by using the triangle area

$$A_{triangle} = \frac{1}{2}|B - A \times C - A| = \pi r^2 = A_{circle}$$
$$r = \sqrt{(\frac{1}{2\pi}|B - A \times C - A|)}$$

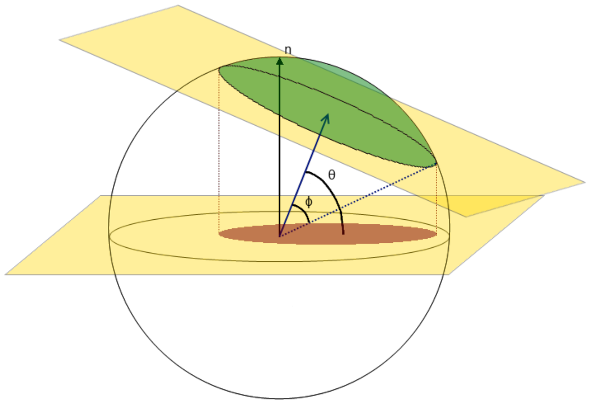

Figure 9: *Circle approximation projected onto tangent plane of $\vec{x}$.*

By finding the radius r and the triangle's centroid c, we get a circle in world space. This alone is not very easy to work with since the AO-values wanted needs to fit in a single texture. It is easier to work with the circle projected onto the sphere surface which can be represented using two angles thanks to the $\vec{x}$->$circle$ vector being orthogonal to the projected triangle plane.

$$\phi = \arctan \frac{r}{|c|}$$
$$\theta = \frac{\pi}{2} - \arccos \frac{c \cdot y}{|c||y|}$$

3.7.3 Finding occluded surface area (Pre-calculated)

Given the sphere projected circle $S(C)$ with circle size angle ϕ we can find the amount of occlusion cast by the approximation circle by projecting the spherical segment surface onto the tangent plane of $\vec{x}$. We thus eliminate the cosine weighted solid angle by changing the integration domain.

$$AO_C = \frac{1}{\pi}\int_{S(C)}(\hat{\omega}\cdot\hat{n})\,d\hat{\omega} = \frac{1}{\pi}\int_{T(S(C))} 1\,d\vec{x}$$

Finding the occlusion from here comes down to finding the projected area in the tangent plane and compare it to the unit circle area.

3.7.4 Performance

Running this algorithm showed time reductions of $20 - 60\%$ depending on scene complexity and the magnitude of the maximum obscurance distance δ. As expected, the difference in performance is greater for more complex scenes and bigger δ-values due to a faster pixel shader program.

3.7.5 Conclusions

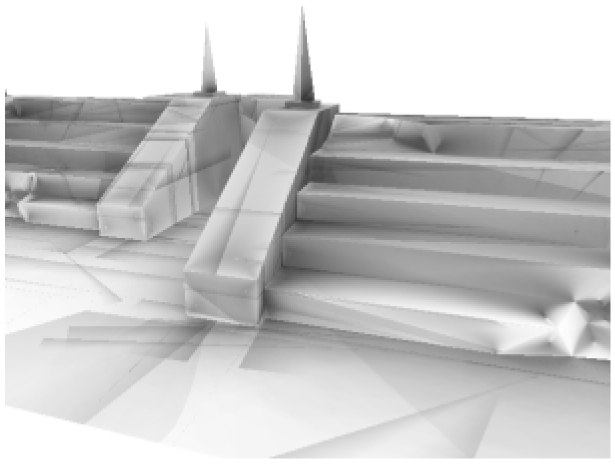

Figure 10: *The results for this technique did not come out very well.*

Although the speed increase of this algorithm is promising, especially for bigger scenes than the relatively small ones used in these tests, the visual results were far from satisfactory as shown in figure

10. The main purpose of the Ambient Occlusion Volume algorithm is to deliver results that are very close to the ground truth and by making too hefty approximations this purpose is easily lost.

3.7.6 Further work

Circles are not very good at approximating triangles. Using eclipses instead would yield more accurate results but is also slightly more computationally heavy.

One such solution would be to store two angles that determine the shape of the ellipse instead of one as in the circle case. This is not the most accurate solution as we still need another angle in order to rotate the ellipse freely, but it could easily be represented by a 3D-texture.

3.8 Pre-calculated Volumes

A natural proposal for making the Ambient Occlusion Volumes algorithm faster is to remove the geometry shader stage by pre-computing the occlusion volumes for all rigid objects on the CPU.

Performance tests showed that this had no effect at all when running on typical laptop GPUs with low fillrates, but that it gave a significant boost in performance on high performance GPUs. It also scales very well with scene complexity if the application can handle the increased memory usage of storing all the extra data.

The visual quality of the images is equal to those rendered by the original AOV algorithm since their pixel shader function and the input it receives are equal for both algorithms.

4 Performance Evaluation

This section provides results from benchmark runs well as a discussion about the work as a whole.

All tests in this section have been run on an NVIDIA Geforce GTX 260 GPU with an Intel Core 2 DUO E8500 clocked from 3.16 to 3.80 GHz and 4GB of physical memory on a Windows 7 64-bit system. The languages of choice for all implementations were C# 3.5 and HLSL using DirectX 10 via the SlimDX API. The most important hardware for all techniques is the graphics card due to the high fill-rate demands of most algorithms.

4.1 Results

All images supplied here were rendered using the baseline algorithm. Only variants that produced results similar to the baseline with no major artifacts are presented. All tests were run using a δ of 2 meters with 1280x720 resolution. No frustum culling or similar optimization techniques were used. Relevant individual settings follow below.

- The limited overdraw method used a cap of 48 overdraws per pixel
- Upsampling used the 2x2 grid mentioned in 3.3.2

Each scene was tested by letting the camera fly around the scene in a pre-determined manner so that the results gave a good overview of the general performance of the algorithm. The camera position and rotation for each frame was set independent of rendering performance which made the algorithms run on equal terms.

The scenes presented below are:

- **Baseline** - The normal AOV algorithm as presented by McGuire.
- **Baked shadows** - Pre-calculated self-occlusion as described in section 3.1.
- **Upsampling (1/2)** - Upsampling as described in 3.3 using a render target texture of half the width and height of the final resolution.
- **Upsampling (1/4)** - Like Upsampling (1/2), but with quarter width and height render target texture.
- **Limited overdraw** - Hard-limiting maximum number of overdraws as described in section 3.5.
- **Pre-calculated volumes** - Pre-calculated volumes as described in section 3.8.

4.1.1 Simple box

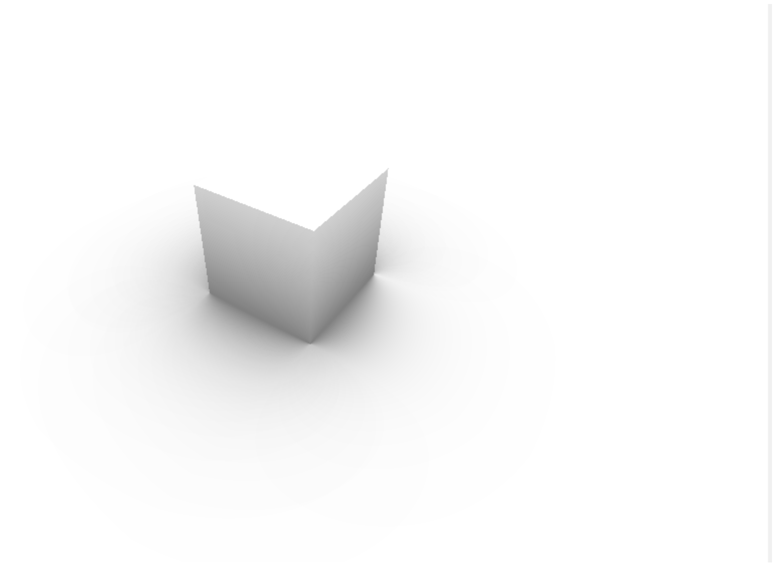

Figure 11: *Simple box scene (14 triangles).*

Technique	**Min. time**	**Avg. time**	**Max. time**
Baseline	0.26	1.84	8.80
Upsampling (1/4)	0.52	1.57	5.47
Baked shadows	0.56	1.64	5.78
Pre-calculated volumes	0.32	1.94	11.61
Upsampling (1/2)	0.56	2.23	5.18
Limited overdraw	0.27	2.73	9.66

Table 1: *Rendering times in milliseconds when rendering the scene in figure 11.*

4.1.2 Multiple boxes

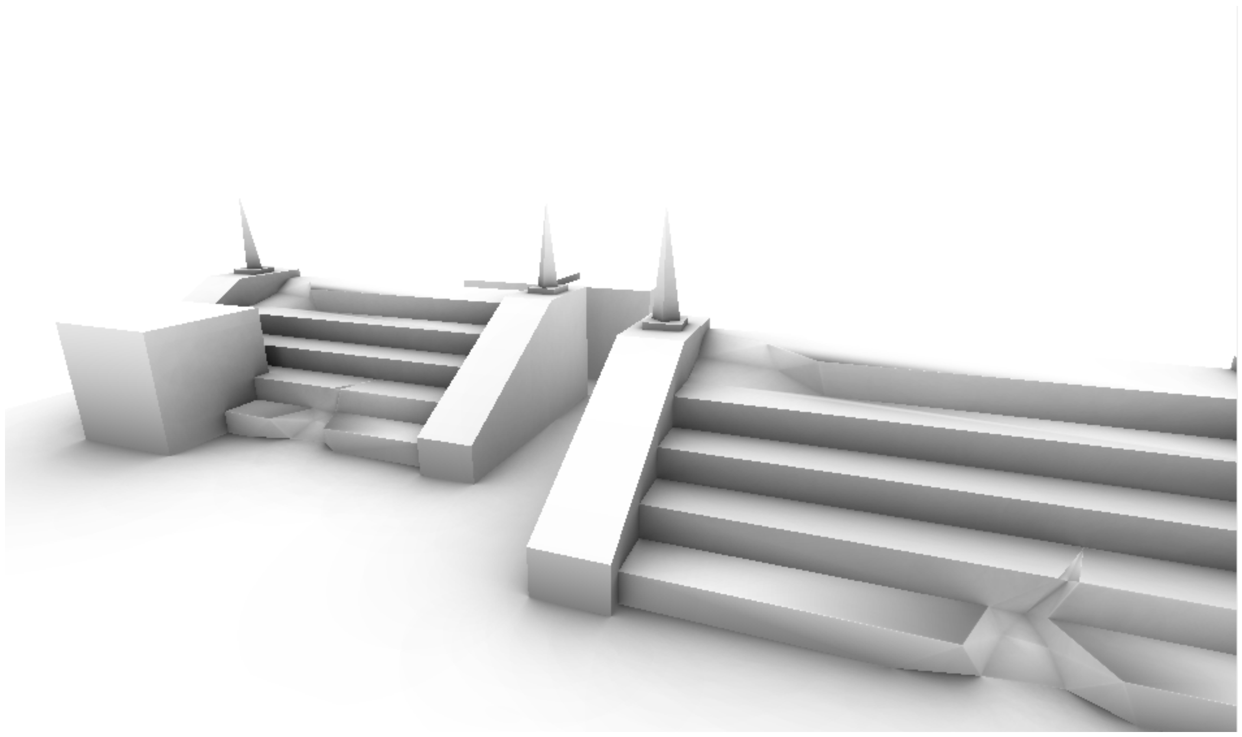

Figure 12: *Multiple boxes scene (278 triangles).*

Technique	**Min. time**	**Avg. time**	**Max. time**
Baseline	0.54	23.69	47.19
Upsampling (1/4)	1.18	4.04	7.83
Pre-calculated volumes	0.54	7.54	24.83
Upsampling (1/2)	1.27	11.2	19.59
Baked shadows	0.88	20.30	39.98
Limited overdraw	0.77	40.02	81.75

Table 2: *Rendering times in milliseconds when rendering the scene in figure 12.*

4.1.3 Stairs

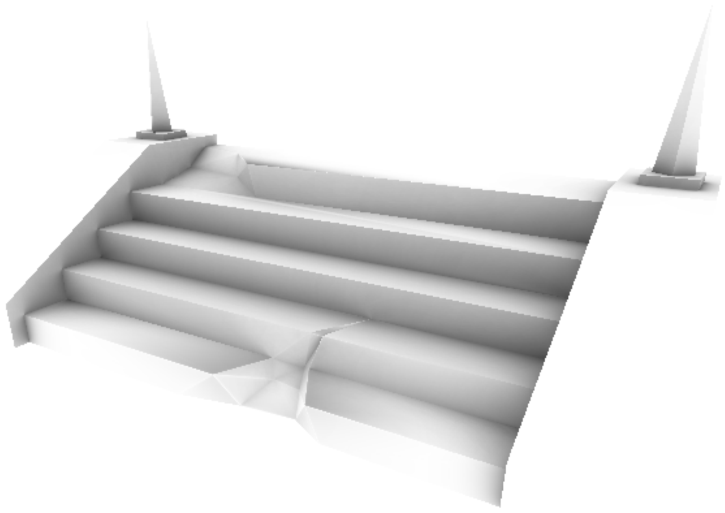

Figure 13: *Stairs scene (120 triangles).*

Technique	**Min. time**	**Avg. time**	**Max. time**
Baseline	0.31	8.71	25.43
Pre-calculated volumes	0.24	1.76	13.75
Upsampling (1/4)	0.48	2.24	7.63
Upsampling (1/2)	0.5	4.93	10.31
Baked shadows	0.89	5.39	12.46
Limited overdraw	0.32	12.79	40.51

Table 3: *Rendering times in milliseconds when rendering the scene in figure 13.*

4.1.4 Indoor scene

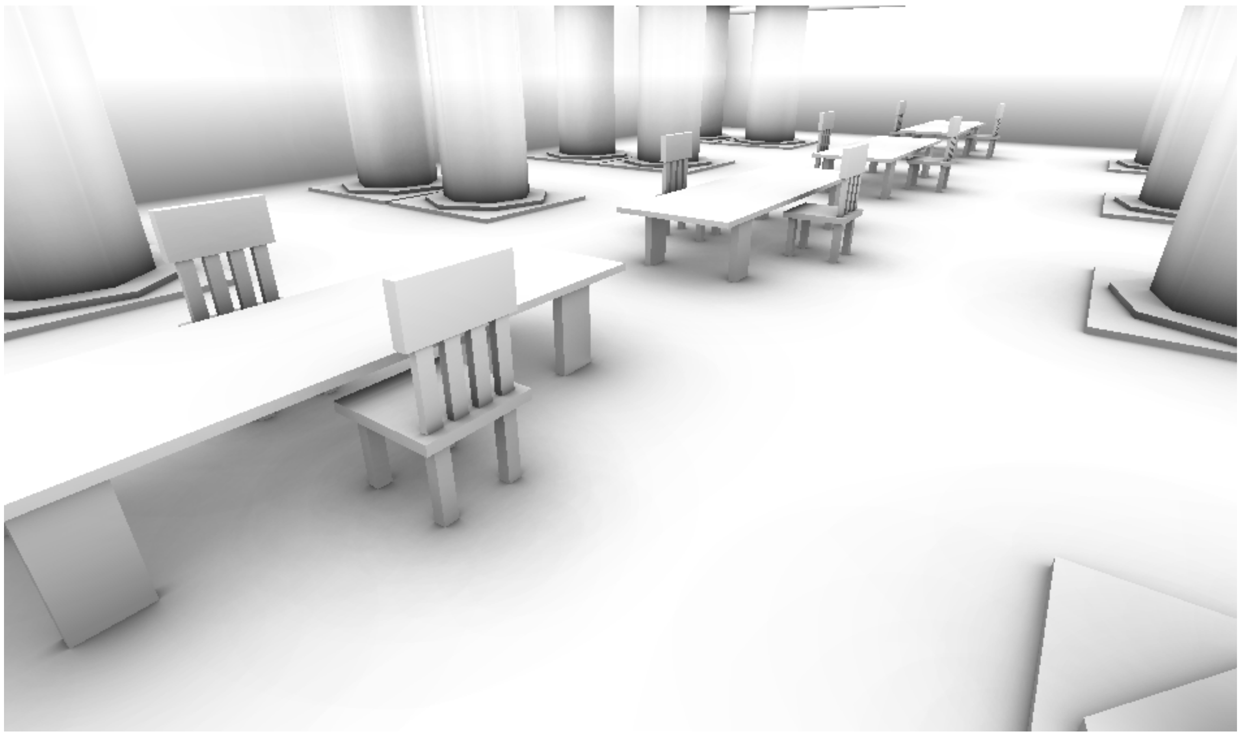

Figure 14: *Indoors scene (3678 triangles).*

Technique	**Min. time**	**Avg. time**	**Max. time**
Baseline	3.46	66.46	93.93
Pre-calculated volumes	2.16	5.11	10.77
Upsampling (1/4)	2.77	10.36	16.06
Upsampling (1/2)	4.71	30.52	97.86
Baked shadows	4.41	61.93	81.53
Limited overdraw	3.50	130.00	163.41

Table 4: *Rendering times in milliseconds when rendering the scene in figure 14.*

4.1.5 Sponza

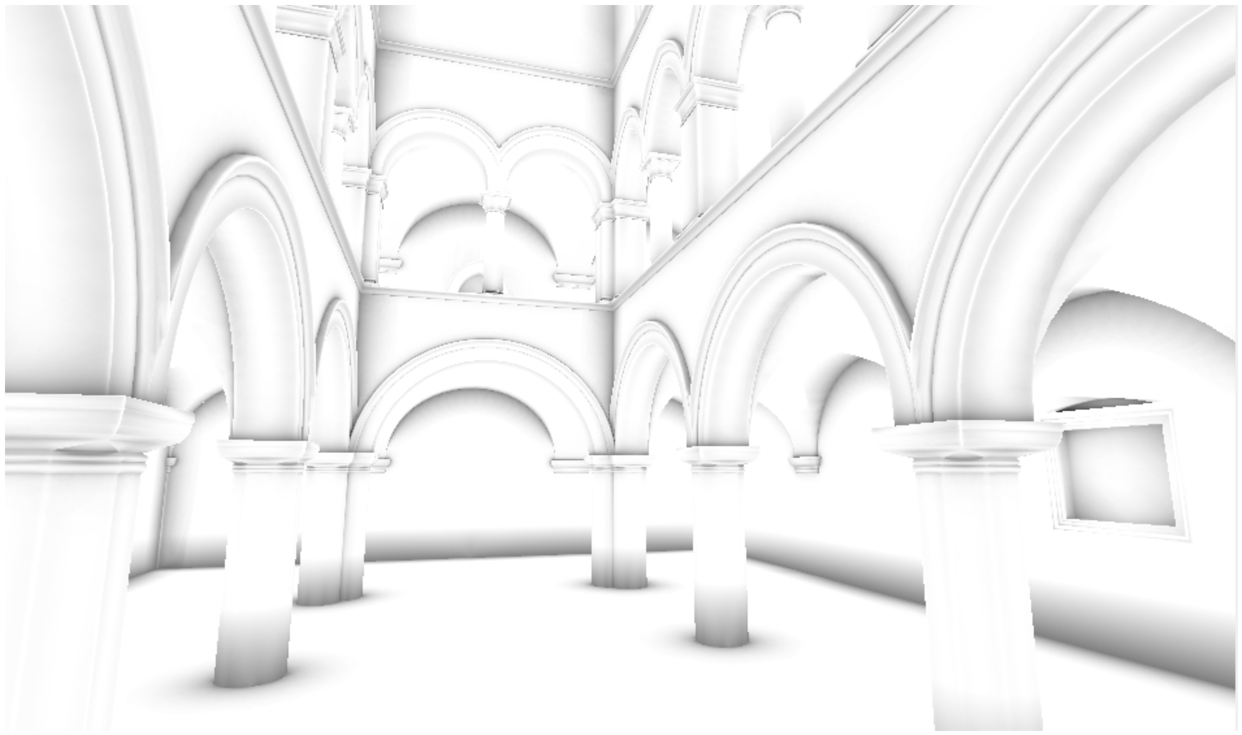

Figure 15: *Sponza scene (66454 triangles).*

Technique	**Min. time**	**Avg. time**	**Max. time**
Baseline	0.24	36.22	81.39
Baked shadows	-	-	-
Upsampling (1/2)	0.45	20.63	41.86
Upsampling (1/4)	0.74	13.48	26.20
Limited overdraw	0.31	60.55	139.94
Pre-calculated volumes	-	-	-

Table 5: *Rendering times in milliseconds when rendering the scene in figure 15.*

This scene was too heavy to load for the pre-calculated shadows method and, due to a problem with the model, the scene would unfortunately not load for the pre-calculated volumes method either.

4.1.6 Summary

Technique	Box	Multiple Boxes	Staircase
Baseline	0.26 / 1.84 / 8.8	0.54 / 23.69 / 47.19	0.31 / 8.71 / 25.43
Baked shadows	0.56 / 1.64 / 5.78	0.88 / 20.3 / 39.98	0.89 / 5.39 / 12.46
Upsampling (1/2)	0.56 / 2.23 / 5.18	1.27 / 11.20 / 19.59	0.50 / 4.93 / 10.31
Upsampling (1/4)	0.52 / 1.57 / 5.47	1.18 / 4.04 / 7.83	0.48 / 2.24 / 7.63
Limited overdraw	0.27 / 2.73 / 9.66	0.77 / 40.02 / 81.75	0.32 / 12.79 / 40.51
Pre-calc volumes	0.32 / 1.94 / 11.61	0.54 / 7.54 / 24.83	0.24 / 1.76 / 13.75

Technique	Indoor	Sponza	
Baseline	3.46 / 66.46 / 93.93	0.24 / 36.22 / 81.39	
Baked shadows	4.41 / 61.93 / 81.53	-	
Upsampling (1/2)	4.71 / 30.52 / 97.86	0.45 / 20.63 / 41.86	
Upsampling (1/4)	2.77 / 10.36 / 16.06	0.74 / 13.48 / 26.20	
Limited overdraw	3.50 / 130.00 / 163.41	0.31 / 60.55 / 139.94	
Pre-calc volumes	2.16 / 5.11 / 10.77	-	

Table 6: *Comparison of time (milliseconds) required to run the benchmark test for the different scenes used in this performance evaluation.*

Based on the above results we can see that all variations perform at least slightly better than the baseline in terms of speed except one, limited overdraw. The maximum number of allowed overdraws needed to be quite high which made the possible gains from the algorithm lose out to its overheads.

Upsampling with 1/4 dimensions on the AOV render target (1/16th of the original area) looks quite cluttered when moving around most scenes, while the 1/2 one fared much better. They both scale pretty well with scene complexity, which makes them viable candidates for real applications.

An algorithm that scaled very well with scene complexity was pre-calculated volumes. Unlike upsampling, however, there is no gain whatsoever from running this algorithm on a GPU with a low fillrate as this quite easily becomes the bottleneck. The resulting image is equal to the one produced by the baseline algorithm.

Baked shadows ran slightly faster than the baseline algorithm, but there was a noticeable drop in shadow quality. This, bundled with the complexity of implementation, makes it hardly worth the effort.

4.2 Discussion

The AOV algorithm has many benefits and is certainly revolutionary in its target area, accurate images at interactive rates. That being said, it still suffers from a few issues such as the high dependency on fillrate and the difficulty in finding a good size on δ. Nevertheless, it is a good thing that it relies so heavily on the GPU since this area currently is advancing at a fast pace, which might mean the algorithm could be used for games in a couple of years.

Being an analytic solution it ensures precise and smooth results with little artifacts. The algorithm does not have any problems dealing with non-rigid objects either which makes it good for 3D-modelling, which is its main target application.

This thesis covers a variety of propositions for improvements; some mentioned by McGuire in his paper and some that was formed during this thesis work. The most applicable ones of those presented in this paper seems to be pre-calculated volumes and upsampling, both coming with their share of downsides and both presented in the original paper. The first can become a bit heavy on memory usage, while the second skimps on image quality.

None of the fresh attempts at improving the algorithm were able to make it all the way, though some were cutting it close. It is possible that some might end up with better results with slight modifications, but there was not enough time allotted in this thesis work for further exploration.

5 Conclusions

5.1 Future work

There are still some features and variants that are yet to be explored. Some are mentioned in the original paper by McGuire, but were not used in the tests of this report.

The best way to improve the algorithm seems to be to limit the amount of rasterized pixels, so that as few unnecessary computations are carried out as possible. One way make the occlusion volumes tighter is to implement a dynamic δ that varies based on the surrounding geometry and the polygon size. However, such an implementation is non-trivial.

The original paper uses a 1D-texture compensation map to mitigate the overshadowing effects that can occur in geometrically dense areas. Doing this can increase the visual results with little performance overhead.

Another method mentioned by McGuire is to represent the geometry with quads instead of triangles. According to the paper this resulted in a noticeable speedup. Unfortunately, the OpenGL and DirectX geometry shaders are unable to output quadrilateral lists which complicates the implementation.

5.2 Conclusion

This thesis examines the Ambient Occlusion Volumes algorithm and presents several attempts of improving it. No better solutions than what was presented in the original paper were found, but the realized attempts might provide useful information or spring new ideas to others exploring the same area.

The algorithm is currently not sufficient for modern games, it fulfills the requirements of 3D-modelling which is the intended usage. Nevertheless, due to the current fast advances in GPU performance, it is far from impossible that we will be seeing the algorithm used in games a few years from now.

References

[Bavoil and Sainz, 2009] Bavoil, L. and Sainz, M. (2009). Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH '09: SIGGRAPH 2009: Talks*, pages 1–1, New York, NY, USA. ACM.

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

[Beam, 2003] Beam, J. (2003). Dynamic lightmaps in opengl. `http://joshbeam.com/articles/dynamic_lightmaps_in_opengl/`.

[Boubekeur and Schlick, 2007] Boubekeur, T. and Schlick, C. (2007). *GPU Gems 3*. Addison-Wesley.

[Evans, 2006] Evans, A. (2006). Fast approximations for global illumination on dynamic scenes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 153–171, New York, NY, USA. ACM.

[Grossman and Dally, 1998] Grossman, J. P. and Dally, W. J. (1998). Point sample rendering. In *Rendering Techniques*, pages 181–192.

[Hr?dek et al., 2003] Hr?dek, J., Kuchar, M., and Skala, V. (2003). Hash functions and triangular mesh reconstruction. *Computers & Geosciences*, 29(6):741 – 751.

[Kontkanen and Laine, 2005] Kontkanen, J. and Laine, S. (2005). Ambient occlusion fields. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, pages 41–48. ACM Press.

[Lei Yang,] Lei Yang, Pedro V. Sander, J. L.

[Malmer et al., 2007] Malmer, M., Malmer, F., Assarsson, U., and Holzschuch, N. (2007). Fast precomputed ambient occlusion for proximity shadows.

[Matt Pharr, 2005] Matt Pharr, R. F. (2005). *GPU Gems 2*. Addison-Wesley Professional.

[McGuire, 2009] McGuire, M. (2009). Ambient occlusion volumes. Technical Report CSTR200901, Williamstown, MA, USA.

[Mittring, 2007] Mittring, M. (2007). Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 97–121, New York, NY, USA. ACM.

[Reinbothe et al., 2009] Reinbothe, C., Boubekeur, T., and Alexa, M. (2009). Hybrid ambient occlusion. *EUROGRAPHICS 2009 Areas Papers*.

[Ritschel et al., 2009] Ritschel, T., Grosch, T., and Seidel, H.-P. (2009). Approximating dynamic global illumination in image space. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82, New York, NY, USA. ACM.

[Wassenius, 2005] Wassenius, C. (2005). Accelerated ambient occlusion using spatial subdivision structures.