# Real-Time Dynamic Ambient Occlusion

Osiris Pérez
*d1151768*
National Centre for Computer Animation
Bournemouth University

September 5, 2005

# Contents

# 1 Introduction

Recent advancements in graphics hardware have opened the door to a new level of flexibility in real-time rendering techniques. The search for realism in video-games and real-time applications has pushed developers to look beyond traditional methods and into borrowing techniques until now reserved for high-end rendering, animation and visual effects.

This work explores some of the latest approaches to simulate a particular global illumination technique called ambient occlusion and a novel implementation using hierarchical disk-area approximations.

# 2 Ambient Occlusion

In computer graphics, it is possible to categorise most rendering algorithms depending on the way they treat light interaction across the scene. According to this criteria, there are two broad categories of light interaction: local models and global models. Whereas the former only takes into account the amount of light at each point based on local information (grazing angle, eye position, etc.), the later models try to mimic the dynamic relationship between each point and the rest of the scene. This type of model is usually called Global Illumination and it is the focus of this paper, particularly, Ambient Occlusion.

Ambient Occlusion differs from most global illumination techniques in the fact that it does not take light directly into account. Whereas other techniques compute the illumination and/or shadowing values at each point, ambient occlusion just determines accessibility information of a surface based on nearby geometry. Such information can then be used to modulate lighting values resulted from any other technique. Therefore, ambient occlusion is often used in conjunction with other shading algorithms.

Although not an illumination technique per se, ambient occlusion is still considered a special sort of global illumination since, unlike local models, the values for each point are determined in relation to nearby geometry.

The occlusion value for any given point on a surface is proportional to the *solid angle* projected by all geometry over a hemisphere around that point. The general equation for the occlusion term of a point $p$ with normal $n$ is as follows:

$$S(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{p}, \omega) \, |\, \mathbf{n} \cdot \omega |\, d\omega \tag{1}$$

where $\Omega$ is a hemisphere around $p$ and $V(p, \omega)$ is the visibility function returning zero if no geometry is visible in the directon $\omega$ and one otherwise [KL05]. Due to

the nature of digital images it is more useful to have the accessibility term instead. This is easily calculated as:

$$A_{cc}(\mathbf{p}, \mathbf{n}) = 1 - S(\mathbf{p}, \mathbf{n}) \tag{2}$$

This is what is commonly referred to as an "inside-out" approach as the accessibility information is determined from the point, based on its relation to the rest of the scene. The "outside-in" approach determines the accessibility of a whole scene by rendering it from multiple viewpoints around an object [Wik] and averaging the results for each point.

Since most of the diffuse contribution at any given point comes from the visible environment from that point, it is also useful to compute an average accessibility vector or, bent normal. This accessibility-weighted normal can be then used to look up the environment so colour or radiance or light values are gathered from the appropriate direction.

## 2.1 Related Work

Until very recently, most work on ambient occlusion had been focused on high-end rendering for film and general or as an off-line process.

According to [KL05] , the technique itself is a generalisation of the ideas introduced by Zhukov et al. in 1998. However, its widespread use can be attributed to the works of Christensen and Landis at ILM.

This section introduces some of the methods used to implement the aforementioned concepts.

### 2.1.1 Ray-traced Methods

One of the simplest methods to determine the accessibility of every point on a surface is to cast rays in the sphere around it. The amount of rays that hit other objects in the scene will determine the occlusion value for that point [Lan02]. The results are usually stored or "baked" into a texture for later reuse or re-computed every frame for dynamic objects.

Usually the ray-casting part of this method uses a Monte Carlo approach in order to minimise artefacts produced by undersampling. Since the strongest diffuse contribution comes from the general direction of the surface normal, the result is weighted to favour samples that are cast in that direction. Other common technique using blurred texture look-ups to interpolate between samples.

The bent normals are determined by averaging the rays that do not hit any object and they can be stored in the same texture as the occlusion terms by using multiple channels.

Although this "inside-out" method is computationally intensive, it yields excellent results and it is often the preferred method for high-end rendering such as feature film animation and effects.

### 2.1.2 Depth Map Methods

Although the use of depth maps for simulation light phenomena is not new, it can be argued its recent success is attributed to the works of Christophe Hery, Ken McGaugh and Joe Letteri at Weta Digital. In their work, depth maps were used to store accessibility and light transport-like information to be used by custom RenderMan shaders.

Emulating these techniques, [Whi] describes an "outside-in" method consisting on using a hemispherical array of spotlights to generate several depth maps. The scene is rendered from each light's point of view and the resulting depth maps are averaged together using a RenderMan shader. The average shadow value at each point on the surface corresponds to the occlusion term.

Despite the fact that this method uses several rendering passes (one for each light plus the final shading passes) it is faster than a full ray-traced solution and has been implemented with small success in real-time environments.

An obvious approach to implementing this method using a GPU is by using textures to store intermediate results. The advantage of this method is that it can use the hardware's texture filtering capabilities to produce smoother results. However, this suffers from similar limitations as it also involves rendering the scene multiple times in order to generate enough information.

Another implementation of this method renders a separate pass per light, adding the results using a floating-point accumulation buffer. The main limitation of this approach is that the number of passes required to produce visually acceptable results defeats the purpose of the implementation.

Table 2.1.2 shows the results of using such approach on a last-generation GeForce7 GPU.

| **passes** | 32 | 128 | 512 | 2048 |
|---|---|---|---|---|
| **fps** | 9.17 | 2.20 | 0.55 | 0.13 |

Table 1: Number of passes vs. frames per second

Finally, one interesting approach found is using hardware occlusion queries to determine the number of fragments visible at each pass and then weighting these values against the total number of fragments to determine visibility at each point [Coo][1].

[1]Refer to Appendix A for a full listing of this method

### 2.1.3 Occlusion Fields

This novel method described in [KL05] uses a similar approach to the ray-traced methods described in 2.1.1. It approximated the occlusion terms for an object using a hemisphere and encodes this information in a cubemap as a pre-processing step. This information is later used when determining object-object occlusion where the solid angle subtended by the occluder is retrieved from the cubemap at run-time, resulting in a reverse hemicube-like [Wat99, pages 311–314] approximation.

## 2.2 Disk-Area Approximation

This technique, presented by [Bun05] works by treating polygon meshes as a set of disks that can transmit, emit or reflect light and cast shadows on each other. For each vertex in the mesh, a disk with the same position and normal is created. The area of a disk $d_i$ corresponding to a vertex $v_i$ is the sum of the area of each face sharing that vertex divided by the number of vertices, or:

$$A_{d_i} = \frac{1}{n} \sum_{f=1}^{n} A_f \tag{3}$$

The disk information is then stored in a texture maps and used by a custom *Cg* shader to compute the occlusion values for each disk. Figure 1 on the following page shows the relationship between disks[2] .

To calculate the occlusion term for each disk, the solid angle is approximated by the integral of the differential area $dA$ over the cone intersecting the hemisphere around the solid angle subtended by an occluding disk at distance $d$

Given that the solid angle is defined by:

$$w = \frac{A}{r^2} \tag{4}$$

where $r$ is the radius of the hemisphere, the differential area $dA$ can be though of as the area of a infinitesimal square. Therefore, the area of a differential square centred in a point $q$ with spherical coordinates $(r, \theta, \phi)$ can be calculated using the differential arc-lenghts $dl_\theta$ and $dl_\phi$ as:

$$\begin{aligned} dA &= dl_\theta \, dl_\phi \\ &= (r \, d\theta)(r \sin\theta \, d\phi) \\ &= r^2 \sin\theta \, d\phi \, d\theta \end{aligned} \tag{5}$$

[2]Image taken and modified without permission from [Bun05]

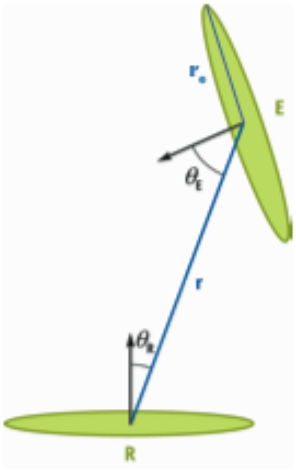


Figure 1: Relationship between emitter and receiver disks

Substituting equation 4 in equation 5 on the previous page gives:

$$d\omega = sin\theta \, d\phi \, d\theta \tag{6}$$

The total area $w$ is given by:

$$\begin{aligned} w &= \int_{\Omega} d\omega \\ &= \int_0^{\theta} \int_0^{2\pi} sin\theta \, d\phi \, d\theta \\ &= 2\pi - 2\pi \cos\theta \\ &= 2\pi \left(1 - \cos\theta\right) \\ &= 2\pi \left(1 - \cos\left(\arctan\left(\frac{r_e}{d}\right)\right)\right) \\ &= 2\pi \left(1 - \frac{1}{\sqrt{1 + \frac{r_e^2}{d^2}}}\right) \end{aligned} \tag{7}$$

Normalising equation 7 to map the resulting value to a $(0, 1)$ range is accomplished by dividing the result by the area of a hemisphere ( $2\pi \, r^2$ ).

The new $w_N$ is then modulated by the grazing angles between the disks, following Lambert's approximation of diffuse light contribution:

$$
\begin{aligned}
S_R &= \cos\theta_R \, \cos\theta_R \, w_N \\
&= \cos\theta_R \, \cos\theta_R \, \frac{1}{r^2} \left( 1 - \frac{1}{\sqrt{1 + \frac{r_e^2}{d^2}}} \right)
\end{aligned} \tag{8}
$$

Note that this equation differs from the one presented in [Bun05, page 225]. However, further analysis of the author's implementation revealed that he is using an equivalent to equation 8 by substituting $r_e$ by $Area/\pi$ according to the disk-area formula and the hemisphere radius is assumed to be 1.

Although disk calculations can be easily performed on the GPU, the real acceleration comes from a hierarchical data structure. Since shadows cast under diffuse lighting are generally soft, those cast from distant objects lack detail. Therefore, it is possible to approximate distant objects by simplified geometry. Neighbouring elements can be grouped and represented by fewer and larger objects, forming a level-of-detail hierarchy of disk approximations instead of polygonal data like previous approaches.

The area of each disk in the hierarchy is simply the sum of its children's and other attributes like positios, normals, etc. are averaged together. These elements are traversed in breadth-first order, according to a distance threshold, considering only the larger element approximations for distant objects. This approach lowers the element-to-element comparison order from $O(n^2)$ to $O(n \log n)$ where $n$ is the number of elements.

# 3 GPU Programming

The GPU, or *Graphics Processor Unit* is the current iteration of the graphics display evolution. Whereas first graphic adapters dealt only with character and image displaying tasks, modern graphics cards posses powerful processor capable of performing operations parallel on huge data sets. Until very recently, these operations were of fixed nature and very specialised but recent advances (and demands) have made it possible to perform more general tasks through programmable pipelines. This section briefly describes current GPU architecture and related technologies.

## 3.1 Basic GPU Computing Concepts

In order to understand the advantages of using modern GPUs to perform tasks, other than traditional graphics processing, it is necessary to adapt many assumptions and techniques found in traditional programming.

One of main differences between current graphics processors and traditional ones, is that the former are designed to perform relatively simple, parallel computations over large datasets. In order to keep operations running in parallel, it is necessary to restrict the types of operations possible. Most of these, apply to gathering, scattering and branching operations.

Since most GPUs lack the concept of all-purpose memory, most of the data handling is done through the use of textures. Until very recently, this limitation hindered the types of data representable and therefore, the type of algorithms as well. However, with new architectures providing integer and floating point textures at various levels of precision, it is now possible to explore new computing techniques. Textures can be used as traditional arrays and recent works have already shown the use of more complex data structures stored as textures.

Another feature missing until very recently is the ability to perform true branching and looping. True hardware-branching became available in latest generation cards and undefined loops are barely supported by cards more than two years old.

## 3.2 Modern GPU Architecture

Due to the very nature of the computer graphics, most graphic pipelines have been structured as a computing states with data flowing as streams between them. Each stage operate over a set of elements such as vertex, triangles or pixels. Although very flexible in terms of graphic operations, fixed pipelines offered little or no programmability to users thus, it was not usable for any task other than processing graphic instructions.

Current GPUs allow the user to program almost any type of functionality at two stages of the graphic pipeline in the form of *vertex programs* and *fragment programs*. These allow the user to write programs on vertex and fragment data respectively. The figure 2 on the next page shows a standard *OpenGL* programmable pipeline [3].

### 3.2.1 The Vertex Processor

The vertex processor operates on incoming vertex values and their associated data. It is intended to perform traditional graphics operations such as: vertex transformation, normal transformation and normalisation, texture coordinate generation and transformation, lighting and colour calculations [Ros04]. Since an image is, in essence, an array of memory, vertex processor are capable of scatter-like operations. Also, recent processors are capable of reading from texture memory,

[3] Image taken without permission from [Owe05]

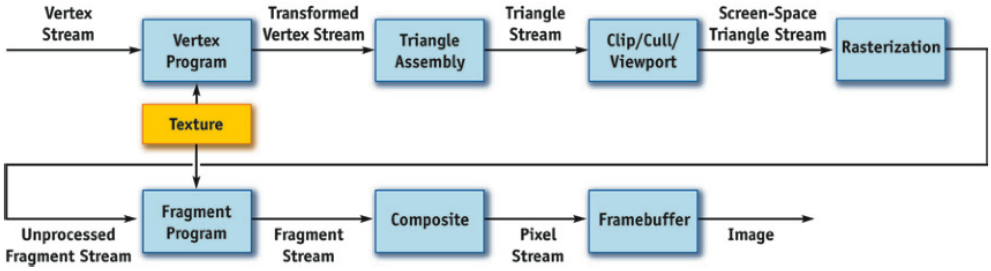


Figure 2: The programmable graphics pipeline.

resulting in a special kind of delayed-gathering operation. We call it delayed because the vertex cannot read information directly from another vertex element but it can read whatever data resulted from a previous computation if it is encoded in texture memory.

### 3.2.2 The Fragment Processor

The fragment processor operates on fragments and their associated data. Some of the operations traditionally associated with fragment shaders are: texture access and application, fog, colour sum and general operations on interpolated values. As with vertex shaders, fragment shaders can be used to perform almost any kind of computation on the GPU. Because of the fragment processor can access texture memory randomly it is very easy to perform gathering operations within a fragment program. In fact, it is not uncommon to use a texture information to perform look-ups on other textures; feature that comes really handy when porting algorithms to this kind of model. As noted in section 4.1.3, this feature was used to perform information gathering operations.

Due the computational frequency of fragment processing the number of fragment processors is higher than the number of vertex processors. Current top-of-the-line cards have about sixteen fragment processors.

# 4 Implementation Details

This project is based on the techniques described in section 2.2. However, the implemented model is slightly different from the one presented by [Bun05]. This section describes these differences and the design decisions that led to them.

## 4.1 General Design

One of the many interesting aspects of program design involving modern GPUs is that, unlike traditional high-level design, it is limited by the limitations of the hardware used. Therefore, one must be aware of the necessary mapping from data structures and algorithms to stream models suitable for the GPU used.

### 4.1.1 Project Goals

Prior to the designing phase, the aim of the project was established. This allowed for a focused, efficient process and clear methodology. The main goals of this project are:

- Reproducing the effect described in section 2
- Implementing the techniques described in section 2.2
- Providing a flexible framework that allows small variations of the same approach

### 4.1.2 Initial Considerations

Once the goals were established, some considerations regarding design, implementation and tools to be used were made.

Since high-level GPU programming is very recent, most of the tools available are extensions to previous technologies or simply going through the initial experimental phase.

Given the nature of the technique to be implemented, it was determined early on that the best platform to implement it would be *Windows*. This allowed the use of the new buffer extensions, not yet present on other platforms. Also, most of the tools for high-level GPU programming are for this platform anyway.

For the shaders, *Cg* was chosen due to a few key reasons. Recent research has focused on this language in particular since it offers the same advantages as any other high-level shading language but it is not bound to any API or platform. Also, it appeared that there was a great deal of documentation available for it. However, it was later found that most of the documentation available is outdated. In fact, some of the language/runtime features used in this project were undocumented/outdated and were discovered by trial and error or reading the source code.

Although some of the key features used in this project are only available for Windows at the moments, it was decided that the rest of the application should be as platform-independent as possible. Therefore, the graphics API chosen was *OpenGL* and the main language *C++*.

### 4.1.3 Data Structure

According to [Bun05], the key to implementing the technique presented in his paper is the use of a hierarchical data structure. Since this structure is to be used to store successive approximations of groups of elements, it should be fairly obvious that it should mimic the functionality of a tree. However, due to the implementation details described in section 3.1, some additional functionality must be provided.

Given that the structure must be stored in a texture map, some of its inherent relational characteristics are lost. Namely, the ability to traverse it in an ordered manner according to certain policy (breadth-first, depth-first, etc.). In order to overcome this limitation, each node in the tree not only maintains references to its children, but also to its siblings. This information is gathered during the mapping step and the indexes of those nodes (children and siblings) are stored in the texture map as $u, v$ coordinates. In essence, each texel $t = (r, g, b, a)$ at a given set of coordinates $(u_t, v_t)$ stores the $u, v$ coordinates for the corresponding node's first child and next sibling, if any. This information was later used to retrieve additional positional data from other textures using the same coordinates.

Also, the data structure should be flexible enough to allow various partitioning schemes [4]. The final structure chosen was a modified variable-breadth KD-Tree. This allows element clustering in groups of arbitrary size.

Unlike traditional spatial tree implementations where the tree is built from top to bottom (or from root to leaves) by recursively partitioning space and determining which elements of the tree at a given level fall into those partitions, the structure used is built from the bottom up. This has a few key advantages over the top-to-bottom design.

The first apparent advantage is that there is no explicit list of primitive elements (vertexes, triangles, etc.) on each node. This allows for these primitive elements to be kept at leaf-level and to build each new level as a layer based on the previous one. Therefore, the tree can be used not only for space partitioning but also for element clustering.

Another advantage of this method is that a tree can have an arbitrary depth since there is no need to start with one single node (the root node) and keep adding levels until the leaves contain just one primitive. It can be argued that this is not a tree but a forest, since the later can always be converted to a tree structure by adding a dummy root node, it is to be referred to as a tree for simplicity. This approach was chosen in order to prevent excessive overhead caused by generating a complete (and large) tree for dense meshes since the overhead of generating and traversing it can out-weight the speed increase over a simple distance-based rejection method.

[4] See section 4.1.4

### 4.1.4 Spatial Partitioning and Clustering

Although not mentioned in [Bun05] , further analysis of the implementation (released a few weeks later) revealed that the tree-like structure was built using texture coordinate coherence for neighbour determination. This approach yields good results for correctly uv-mapped meshes but it can be very fragile.

It was determined that a more general solution was to be found so a new space partitioning algorithm was developed. This algorithm had to allow for small clusters of an arbitrary size, as well as adaptive clustering when the mesh is not evenly tessellated.

The method developed consisted on generating new levels in the form of layers and consisted on two main phases: the partitioning phase and the clustering phase. As described below, the clustering phase requires that a separate list is kept with the elements corresponding to the most recent level of the tree.

First, a node from the current level is selected as a reference. The current level is then partitioned according to the direction of the disks normals. This arbitrary policy can be as general as selecting the normals in the same hemisphere or as strict as selecting just the normals in a subsection of a hemisphere (a cone).

The second phase uses the partition corresponding to the reference (normals facing similar directions) and performs a partial sort of the first $k$ elements according to distance where $k$ is dimension of the KD Tree. This elements are then grouped together under a new element of the level being built and removed from the top level. Algorithm 1 on the following page shows the process of building a new level.

Element-to-element comparison is done from the top level, based on a distance threshold for controlling the level of approximation used during traversal.

### 4.1.5 Data Storage and Transfer

As mentioned in section 4.1.3, disk attributes are to be stored in and indexed by floating-point textures. This presented a couple of challenges: data transfer and data alignment.

The first challenge arose from the necessity of indexing and updating disk attributes. Three textures were used: an index map, a position map and a normal-area map. Each disk must have the same $(u, v)$ coordinates in each map. Since current GPUs do not support more than 4,096 elements in a single array [LKO05], it was necessary to map the disk-tree to a 2-dimensional texture. Assuming the disks have unique, sequential identifying numbers, it is possible to map them to unique 2D addresses in texture space. This information is then uploaded to video memory using a simple copy-to-texture operation. However, the challenging part is computing the ambient occlusion results from these textures.

**Algorithm 1** New Level Creation

$Top' = \emptyset$
**while** $Top \neq \emptyset$ **do**
  $r \, \epsilon \, Top$
  $Top' \leftarrow Top' \cup \{r\}$
  $Children_r = \emptyset$
  $P = \emptyset$
  **for all** $d_i \epsilon \, Top$ **do**
    **if** $N_i \cdot N_r > \delta$ **then**
      $P \leftarrow P \cup \{d_i\}$
      $Top \leftarrow Top - \{d_i\}$
    **end if**
  **end for**
  **for** $0 < i < k$ **do**
    $m = \min(P)$
    $Children_r \leftarrow Children_r \cup \{m\}$
    $P \leftarrow P - \{m\}$
  **end for**
**end while**
$Top \leftarrow Top'$

Although the occlusion terms are computed on a per-vertex basis, a fragment shader was used for this task. The main reason for this decision is based on the way polygonal meshes are drawn. The reader should be familiar with vertex sharing among polygonal faces. This leads to redundancy in the vertex-level computation (at least from a programmer's point of view. Some modern GPUs have specialised hardware to prevent unnecessary transformations but it is not guaranteed to work for every case). Thus, introducing overhead by calculating the occlusion terms more than once per vertex. This is also avoided by mapping the disks to a 2D texture using a 1-to-1 correspondence as described previously.

However, this mapping introduces some complexity in the way information is transferred within the application. Because the occlusion computation is done once the data is sent to the graphics pipeline, it is necessary to read it back in order to perform the final colour calculation. There are many ways to do this. The simplest (and of course, slowest) is reading the results directly from the frame buffer. A more sensible approach is to use a pixel buffer for storing the results and reading back as a texture in another rendering pass.

Pixel buffers are memory extensions that provide DMA-like access from the GPU and the host application as well. By writing directly to texture memory, it is possible to store results for immediate use by the graphics pipeline. This

operation was implemented as separate render pass, using a pbuffer render-to-texture extension. What this does is basically, redirect the output of a graphic operation straight to texture memory instead of the frame buffer. By doing this, it is possible to read this information back as a texture and use it in subsequent passes. However, this requires that the data is perfectly aligned so each pixel on screen (and therefore, each disk) corresponds a texel in texture space. This is done by rendering a rectangle containing all disks and aligning the viewport so that the corners of the rectangle match the ones on the viewport and disabling any additional transformations.

## 4.2 Current Implementation

The current iteration of this project is implemented as a standalone *C++* program. The mesh data is loaded from an *.obj* file and it is then passed on to the tree. Then, the building phase starts and the new levels are formed from disk clusters.

The mapping phase starts by copying the relevant disk-node data to a buffer and then uploading it to texture memory using three separate 4-channel floating-point textures. These are used by the *Cg* shader *dao.cg* to calculate the occlusion terms for all vertexes which are in turn rendered as a viewport-aligned *GL Quad* to get the pixel-texel correspondence.

This viewport-aligned texture is also a 4-channel floating-point texture containing the bent normals and occlusion value.

The final rendering pass passes all the face-vertex data using a vertex array and uses another *Cg* shader to read back the occlusion terms and bent normals from texture memory. Although this shader currently displays just the occlusion values as grayscale values, it can be easily modified to incorporate direct lighting and colouring as well.

Although not perfect, the current implementation shows great potential. The following section explains most problems encountered and some of the solutions implemented.

### 4.2.1 Problems encountered

Since shader programming is still relatively new, most tools lack the functionality high-level programmers are used to. Although the main application was programmed using *C++*, the *OpenGL* and *Cg* parts were difficult, if not impossible to debug completely.

Given the nature of the new programmable pipelines, most *OpenGL* debuggers (more like state-viewers) were not able to keep track of system calls and state changes beyond certain point. Also, there is no *Cg* debugger currently available.

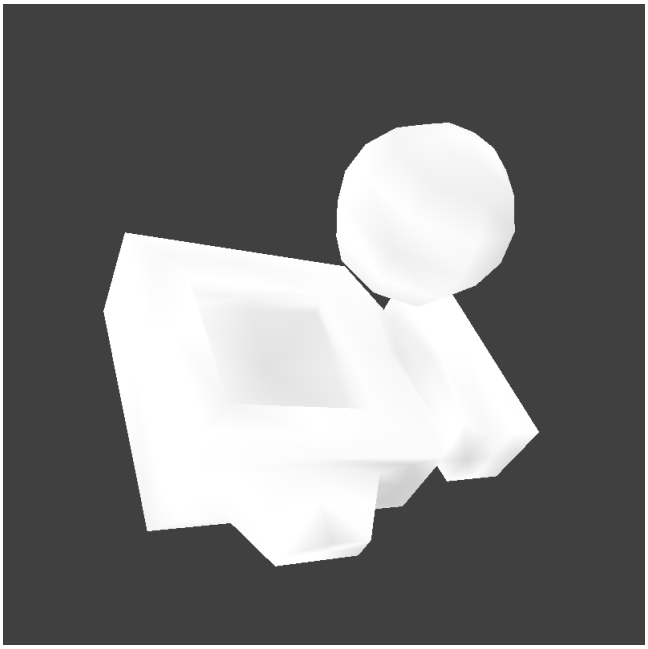

Figure 3: Simple shape rendered using disk-area approximation

Some attempts were made on using a commercial *GLSL* debugger on *OpenGL*-compliant compiled *Cg* code but without success.

Some bugs in the libraries used required code modification/re-writing and a couple application modifications as the tree alignment was heavily impacted by small errors in the loading routines. Also, the render-to-texture extension abstraction library used presented minor problems.

The overall performance seems to be acceptable and, although the visual appearance is sometimes close to other ambient occlusion techniques, it is plagued by artefacts beyond the normal level of conformity. Figure 3 shows a simple shape rendered with the current version.

Further experimentation helped determined that the techniques described in section 2.2 are highly dependant on tessellation and mesh relation. Fortunately, the solution was flexible enough to allow most of the parameters (arbitrarily chosen on [Bun05]) and tweaking can be done on a per-mesh basis. Also, Bunnel's implementation uses a custom mesh subdividing library which, based on other nVidia demos, is suspected to be peforming adaptive subdivision as well.

Figures 4 on the following page show the same model, rendered using different distance thresholds and weights.

It was believed that the the visual anomalies were caused by the clustering algorithm itself. However, during the testing phase, it was possible to bypass

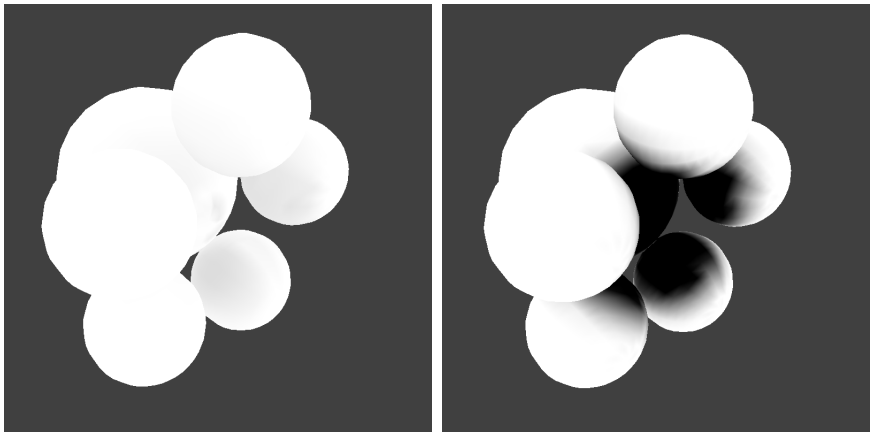

Figure 4: Different parameters affect the final image

the algorithm by not building a hierarchy at all during execution and performing full disk-to-disk comparisons. The results were similar to the ones from the full algorithm.

Similar techniques were applied to the *Cg* shader itself and, although no solution for the artefacts was immediately apparent, it was determined that there is a very high relation between the grazing angle and the distance threshold factor. However, since both these variables are also dependant on the object's tessellation no general solution was found.

# 5 Conclusion

Although the current implementation of this work is not complete, it shows great potential to become a flexible approach to the techniques described here.

The latest graphics hardware generation is starting to show the potential of GPU-accelerated algorithms. Not only is hardware becoming faster and more flexible, but the tools are slowly maturing into fully-fledged programming environments and solutions. Future revisions of current languages are incorporating better abstraction techniques and more complete programming models.

## 5.1 Further Work

Further experimentation should be done in order to determine the cause of the artefacts. Also, it is believed that this will also provide more insight into technique; possibly gaining from a better implementation as well.

Although some of the mechanisms are in place, adaptable clustering techniques are still to be implemented. This could lead to speed improvements as well as artefact reduction. Since there is a direct relation between the disk-area and overlapping distance with other disks, it is possible to tie the distance threshold to the area. Thus, performing adaptive clustering and traversal of the tree.

One of the most interesting extensions is using the same structure for more complex phenomena such as light transfer and bouncing. The later is briefly mentioned in [Bun05] as an addition to the model presented.

By treating disks as an energy transporting element, it is possible to simulate the effects of light travelling and scattering inside a mesh. Since the energy hitting the front face of a disk can bounce of the surface or be absorbed, the later can be thought of as the remainder energy released from the back of the disk. This can be later used for simulating translucence and scattering effects.

# References


[BP04] Ian Buck and Tim Pucell. A toolkit for computation on gpus. In Fernando [Fer04].

[Buc05] Ian Buck. Taking the plunge into gpu computing. In Pharr [Pha05].

[Bun05] Michael Bunnel. *Dynamic Ambient Occlusion and Indirect Lighting*, chapter 14, pages 223–233. In Pharr [Pha05], 2005.

[cg:04] *Cg Toolkit 1.4 User's Manual: A Developer's Guide to Programmable Graphics*. nVIDIA, 2004.

[Coo] Greg Coombe. Assorted notes about ambient occlusion. `http://www.cs.unc.edu/~coombe/research/ao/`.

[Fer04] Randima Fernando, editor. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison Wesley, 2003.

[For] Od Force. Fast gi anyone?, interesting new papers. `http://odforce.net/forum/index.php?showtopic=2876&st=0`.

[Gre05] Simon Green. Gpu programming exposed: The naked truth behind nvidia's demos. In *GDCE 2005*, 2005.

[Har05] Mark Harris. Mapping computational concepts to gpus. In Pharr [Pha05].

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library : A Tutorial and Reference*. Addison Wesley, 1999.

[Kil05] Emmett Kilgariff. The geforce 6 series gpu architecture. In Pharr [Pha05].

[KL05] Janne Kontkanen and Samuli Laine. Ambient occlusion fields. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 41–48, New York, NY, USA, 2005. ACM Press.

[Lan02] Hayden Landis. Production-ready global illumination. 2002.

[LKO05] Aaron Lefohn, Joe Kniss, and John Owens. *Implementing Efficient Parallel Data Structures on GPUs*, pages 521–525. In Pharr [Pha05], 2005.

[MDS01] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley, 2nd edition, 2001.

[nvi05] *NVIDIA GPU Programming Guide 2.4.0.* nVIDIA, 2005.

[Owe05] John Owens. *Streaming Architectures and Technology Trends*, chapter 29, pages 457–470. In Pharr [Pha05], 2005.

[Par01] Richard Parent. *Computer Animation : Algorithms and Techniques.* Morgan Kaufmann, 2001.

[PG04] Matt Pharr and Simon Green. *Ambient Occlusion*, chapter 17, pages 279–292. In Fernando [Fer04], 2004.

[Pha05] Matt Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley, 2005.

[Ros04] Randi Rost. *OpenGL Shading Language.* Addison-Wesley, 2004.

[Shr99] Dave Shreiner. *OpenGL Reference Manual : The Official Reference Document to OpenGL, Version 1.2.* Addison-Wesley, 1999.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, New York, NY, USA, 2002. ACM Press.

[Str00] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 3rd edition, 2000.

[TL04] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. *ACM Trans. Graph.*, 23(3):469–476, 2004.

[Wat99] Alan H. Watt. *3D Computer Graphics.* Addison Wesley, 3rd edition, 1999.

[Whi] Andrew Whitehurst. Depth map based ambient occlusion lighting. http://www.andrew-whitehurst.net/amb_occlude.html.

[Wik] Wikipedia. Ambient occlusion. http://en.wikipedia.org/wiki/Ambient_occlusion.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide : the Official Guide to Learning OpenGL, version 1.2*. Addison-Wesley, 3rd edition, 1999.

# A The Occlusion Queries Method

```
for( some number K camera samples )
     place a camera at a random point on the sphere surrounding the object
     // First pass - fill the depth buffer
     glClear();
     glEnable( GL_depth_compare );
     for( each triangle i in model )
          render( tri[i] );
     end

     // Second pass - render into the depth buffer w/occlusion queries.
     // This will give the number of unoccluded fragments
     for( each triangle i in model )
          glBeginOcclusionQuery( i );
          render( tri[i] );
          glEndOcclusionQuery();
     end
     // Get occlusion queries with depth testing
     for( each triangle i in model )
         triangle[i]->TotalFragments = glGetOcclusionQuery(i);
     end

     // Third pass - render without depth testing.
     // This gives the total number of fragments (unoccluded + occluded)
     glDisable( GL_depth_compare );
     for( each triangle i in model )
          glBeginOcclusionQuery( i );
          render( tri[i] );
          glEndOcclusionQuery();
     end
     // Get occlusion queries without depth testing
     for( each triangle i in model )
         tri[i]->VisibleFragments = glGetOcclusionQuery(i);
         tri[i]->AmbientOcclusion += tri[i]->VisibleFragments /
          (tri[i]->TotalFragments * K);
     end
end
```