Comenius University

Faculty of Mathematics, Physics and Informatics

Department of Computer Science

Filip Zigo

Real-time global illumination of dynamic scenes

Master’s Thesis

Advisor: Doc. RNDr. Roman Ďurikovič, PhD

BRATISLAVA 2008

By this, I declare that I wrote this thesis by oneself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

Bratislava, May 2008 Filip Zigo

Acknowledgements

I would like to thank Roman Ďurikovič, my diploma supervisor, for his time, advices and support.

Abstract

Názov: Globálne osvetlenie dynamických scén v reálnom čase
Autor: Filip Zigo
Dipl. vedúci: Doc. RNDr. Roman Ďurikovič

Globálne osvetlenie predstavuje v počítačovej grafike stále výzvu.
Táto práca sa zaoberá možnosťami interaktívneho výpočtu globálneho osvetlenia.
Úvod práce je venovaný problému globálneho osvetlenia a formulovaný ako úkol nájsť riešenie zobrazovacej rovnice. Najskôr predstavíme klasické prístupy známe z fotorealistického zobrazovania a následne analyzujeme algoritmy aproximujúce zobrazovaciu rovnicu v reálnom čase. Porovnáme navzájom jednotlivé algoritmy. Na implementáciu vyberieme vhodný algoritmus bežiaci na GPU na výpočet globálneho osvetlenia v komplexných dynamických scénach bežiaci v reálnom čase.

Kľúčové slová: global illumination, real-time rendering, complex scenes, dynamic scenes, gpu programming

Abstract

Title: Real-time global illumination of dynamic scenes
Author: Filip Zigo
Advisor: Doc. RNDr. Roman Ďurikovič

Real-time global illumination represents still challenge in computer graphics.
This work deals with possibilities of interactive calculation of global illumination.
Introduction is dedicated to problem of global illumination and formulated as task of searching solution of rendering equation. First we present traditional approaches known from photorealistic rendering and we subsequently analyze algorithms approximating rendering equation in real-time. For implementation we choose algorithm implemented on gpu for calculation of global illumination in complex dynamic scenes running real-time.

Keywords: global illumination, real-time rendering, complex scenes, dynamic scenes, gpu programming

Contents

List of Figures

1. Introduction

Main objective of global illumination and maybe whole computer graphic is finding method, which will be simulate all optical phenomena in real-time. Global character of correlation between objects formally describes rendering equation. Solution of rendering equation is impossible in general case. Therefore rendering methods are approximation of analytical solution.

1.1 Local versus global illumination

A lot of today's applications use rendering algorithms working only with local lighting. Term local means that lighting of object is affected only with properties its surface and lighting from direct light sources. Indirect lighting acquired from environmental elements is ignored, what has negative consequences on authenticity of generated pictures. Therefore no interesting effects appear in final image e.g. soft indirect lighting from lamp, color interaction between different colored walls – *color bleeding* and effect created by rays passing through glass or semi-transparent objects – *subsurface scattering*. Responsibility for realistic look of scene lies on author of the scene, which must properly arrange lights. Algorithms for global illumination take account of impact of indirect lighting. Single surfaces are considered as light sources and all light bounces are simulated. Problem is not only internal complexity and algorithm difficulty but also computational complexity; therefore there are preferred algorithms for local lighting too. But this situation will change with growing power of today's computers.

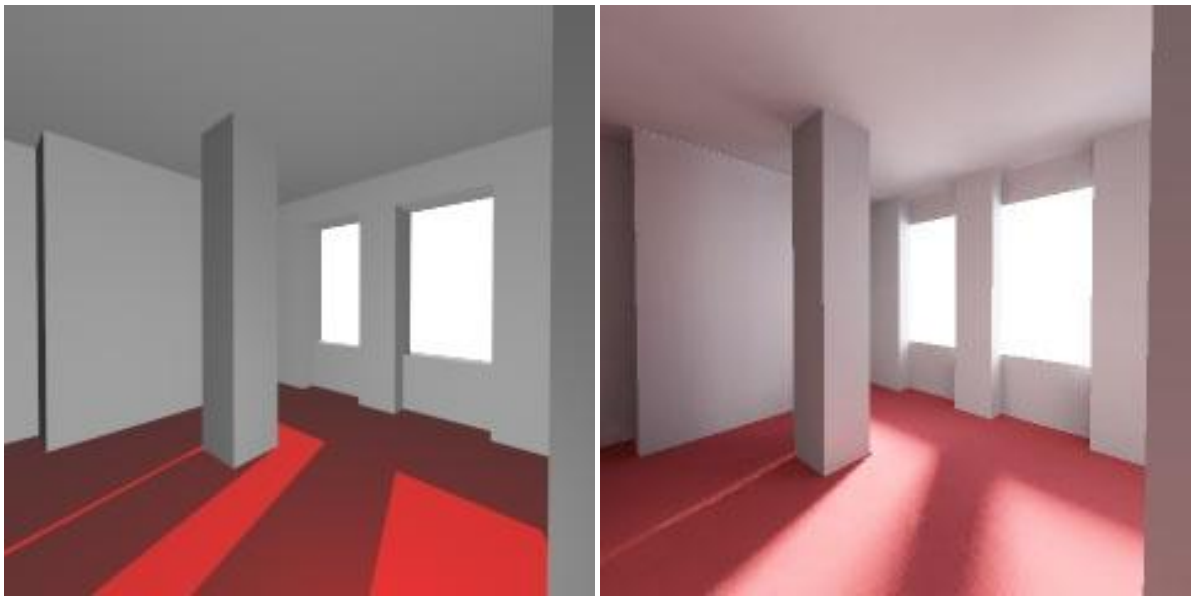


Figure 1: Comparison local and global lighting. *

* http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm

1.2 Rendering Equation

The rendering equation models the equilibrium of the flow of light in the scene.
It can be used to determine incoming radiance into each point on rendering plane of virtual camera.

$$L(p \rightarrow d) = L_e\,(p \rightarrow d) + \int_{\Omega} f_r(p, \mathrm{s} \;\rightarrow \mathrm{d})\mathrm{L_i}(\mathrm{p} \leftarrow \mathrm{s})\mathrm{H}_{\mathrm{N_p}}(-\mathrm{s})\,ds$$

- $L_e\,(p \rightarrow d)$ - emitted radiance from p in direction d
- $\mathrm{L_i}(\mathrm{p} \leftarrow \mathrm{s})$ - incoming radiance from direction s
- $f_r(p, s \;\rightarrow \mathrm{d})$ - BRDF
- $\mathrm{H}_{\mathrm{N_p}}(-\mathrm{s})$ - comes from lamberts law.
- Lamberts law - cosine between normal and –s = $dot(\mathrm{N_p}, -\mathrm{s})$
- Ω - all incoming directions to p

The first term is the radiance emitted directly from the point in the given direction. This is followed by an integral over the hemisphere around the point, where s is used to denote a direction on this hemisphere. First factor is BRDF of surface at point P. BRDF is 4D function that models what percent of light for some input direction (s) leaves in some outgoing direction. The final term is the cosine term that comes from lamberts law – due to projected area.

2. Traditional Approaches to Realistic Image Synthesis

2.1 Ray Tracing

In [Joz02] Ray tracing is a rendering technique that is based on the idea that the only important light rays in a model are the ones that will eventually been seen by the observer, or eye. In order to render an image then, one should trace a ray from the eye through a pixel in the image, and find the first surface the ray intersects. Once this intersection point is found, a new ray, called a *shadow ray*, is traced from the intersection point to the light source. If this shadow ray first hits another object before reaching the light source, then the surface is shaded at the intersection point. If not, the light source is illuminating the surface at the intersection point (Figure 2). If the surface is a *diffuse* surface, meaning neither shiny nor transparent, then the color of the pixel can be calculated immediately. If, on the other hand, the surface is *specular*, a *reflected ray* or *transmitted ray* should be generated. This ray is then recursively sent through the ray tracing algorithm to find the color seen through or reflected by the surface. This process is repeated for every pixel in the image (or several times per pixel), until a complete image is created [Rev04]. The color of each pixel is calculated by a combination of the amount of direct light seen at the intersection point of the ray and the surface, the surface color itself, and if the surface is specular, the color of the surface seen through a reflected or transmitted ray. To calculate the color *C* of a diffuse surface with color *S*, normal $\vec{n}$, and at position $\vec{s}$ under a light with color *L* and position $\vec{l}$, the following is performed:

$$C = L \cdot S(\vec{n} \cdot (\vec{l} - \vec{s}))$$

To calculate the reflected direction $\vec{r}$ of an incoming direction $\vec{d}$ at a specular surface with normal $\vec{n}$, the following is performed:

$$\vec{r} = 2(\vec{n} \cdot \vec{d})\vec{n} - \vec{d}$$

To calculate the direction of a transmitted ray $\vec{t}$ of an incoming ray with direction $\vec{d}$ at a specular surface with normal $\vec{n}$ the following is performed: (*Note:* the ray is traveling from a medium with refractive index m_1 to a medium with refractive index m_2):

$$\vec{t} = \frac{m_1(\vec{d} - \vec{n})(\vec{d} * \vec{n})}{m_2} - \vec{n}\sqrt{1 - \frac{m_1^2(1 - (\vec{d} * \vec{n})^2\theta}{m_2^2}}$$

There is some disagreement between the terms “ray tracing” and “backward ray tracing”. Some think “ray tracing” should be the process of tracing a ray from the light to the eye, whereas other think it should the process of tracing a ray from the eye to the light. For the rest of this paper, Glassner’s perspective will be used and “ray tracing” will be defined as the rendering method suggests, tracing a ray from the eye to the light, and “backwards ray racing” will be defined as tracing a ray from the light to the eye [Gla89].

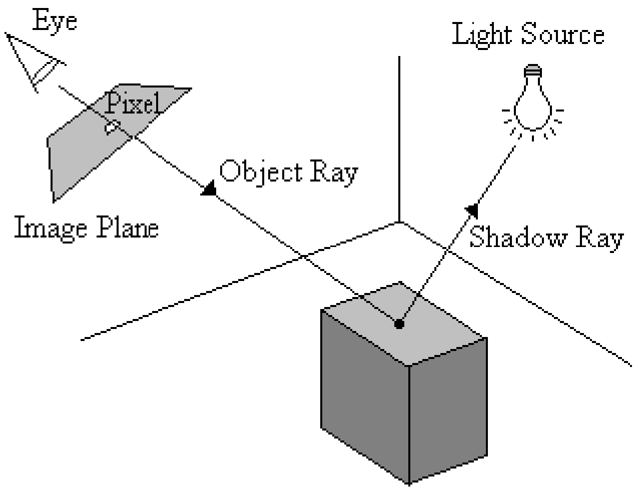


Figure 2: The ray tracing concept. [Joz02]

2.2 Radiosity

Radiosity is the rendering technique takes into account only the diffuse reflections between surfaces. In this case can be rendering equation simplified to diffuse rendering equation:

$$B(x) = B_e(x) + \frac{\rho(x)}{\pi}\int_A B(y)G(x,y)V(x,y)dy$$

Where G(x,y) is geometry term $\frac{\lfloor x\to yn_x \rfloor \lfloor y\to xn_y \rfloor}{r^2 xy}$ and V(x,y) is visibility function, being one when point x is visible from point y, and zero otherwise. By reformulating diffuse rendering equation we can get radiosity rendering equation:

$$B_i = E_i \, \rho_i \sum_{j=1}^{N} B_j F_{ij}$$

F_{ij} represents form factors between two patches and mean the percentage of the outgoing light from the first patch that is visible from the second patch.

Process of radiosity method consists from:

1. Subdividing surfaces into smaller regions also known as patches.
2. Calculating form factor for each couple of patches.
3. Solving linear equation system
4. Display resulted solution from camera view

Radiosity has problems with specular reflections and transparency that were easily done using ray tracing. So next logical step for researchers was to combine radiosity and ray tracing techniques into one single solution.

2.3 Photon Mapping

We summarize this algorithm according to Jozwowski's work [Joz02], Marlon's book [Mar02] and Larsen's work [Lar04]. This algorithm was created in 1994 by Henrik Wann Jensen as part of his Ph.D. dissertation [Jen01]. This algorithm is two-pass and it achieves everything that radiosity provided but with the speed, simplicity, and surface generality of ray tracing.

The observer and the light sources are the two major elements we must take into account when rendering, because they both reveal very important information about the flow of light. The scene is rendered using the eyes of the observer and the lighting comes from the light sources. We must utilize both of these aspects to calculate the lighting correctly. The light must be scattered (find the irradiance) and then observed (find the radiance).

The premise behind his method is simple; in the first pass (*photon tracing*) one simply shoots photons from light the sources into the scene and keeps track of the energy they give to encountered surfaces. Whenever a photon hits a diffuse surface, it is stored in a data structure called the *global photon map*. . The second pass (photon rendering) then uses ordinary ray tracing in conjunction with the photons stored in the global photon map to capture all of the illumination types currently needed in realistic image synthesis. The result of this combination of both specular and diffuse reflections is a full global illumination model that can capture many of the surface interactions that occur in real life.

Dividing the Incoming Radiance

The basic principle of photon mapping is to divide the incoming radiance into a number of components which can be handled individually. The incoming radiance at a sample point can be divided into three components

$L_i(x,w)=L_{i,l}(x,w) + L_{i,c}(x,w)+L_{i,d}(x,w)$

where

- $L_{i,l}$ is the direct light , e.g. L(D|S)E
- $L_{i,c}$ is the caustics, e.g. L(S+)DE
- $L_{i,d}$ is the indirect light,e.g. L(D|S)*D(S|D)+E

The indirect and caustic illumination are calculated from the photon maps by using a density estimation technique. In the following, we will give a more detailed description of this process.

2.3.1 The First Pass: Creating the Global Photon Map

As previously stated, the first pass of the algorithm is to shoot photons from the light sources and then store their progress in a data structure. In order to shoot the photons we must know what type of light source we are dealing with. There are several types of light sources as shown below [Jen01].

- *Point Light*: A single coordinate in 3-dimensional space where photons can be shot by randomly selecting any point on the unit sphere and shooting them in that direction. Jensen suggests using rejection sampling to constrain random vectors inside the unit cube to those inside the unit sphere to randomly generate point light photon emission.

- *Square Light*: This is, in general, the most popular type of light used in most rendering techniques. It looks realistic, and is easy to implement. A square light is simply a square surface that can emit photons from any point and in any direction toward the normal. There are also several techniques using square lights to allow for soft shadow generation.
- *Complex Light*: These are lights with an arbitrary shape that emit photons at different proportions in different locations.

Of course, if there are multiple lights in a scene, a light must be chosen before a photon can be shot. However, one cannot simply choose a random light. The probability of a light being selected must be proportional to the fraction of the light's power over the total amount of power from all lights in the scene [Jen01]. Once the origin of the photon and its outgoing direction vector are known, the photon can finally be shot. Using much the same algorithm as traditional ray tracing, photon tracing finds the first surface that the photon would hit and proceeds to update the global photon map in one of the following ways depending on the type of surface intersected:

- If the photon hits a diffuse surface, store the photon's energy and incident direction into the global photon map.
- If the photon hits a specular surface, do nothing to the global photon map. The global photon map will only consider photons that hit diffuse surfaces.
- If the surface is partially diffuse and specular, use what is called *Russian roulette* to decide if at this particular instance the surface should be considered diffuse or specular.

In any case, whether the photon is absorbed or reflected by the surface must now be decided. If the photon is absorbed, this particular photon's life is ended here. If the photon is reflected (or transmitted) the reflected ray must be calculated the process repeated as if the photon started from the intersection point. Diffuse and specular surfaces do not reflect photons in the same way, and these differences must be accounted for. Specular surfaces can be assumed to reflect perfectly by the relation: $r=2(n*d)*n-d$, where je r is reflected direction, n is normal and d is incoming direction. Diffuse surfaces can reflect in any direction. This direction however is proportional to the cosine of the angle between the perfectly reflected direction and the actual reflected direction. To create the random reflection direction two random variables, $\xi 1, \xi 2 \in [0, 1]$ are needed. The reflected direction *r* is then given by [Jen01]:

$$r = (\theta, \phi) = \cos^{-1}(\sqrt{\xi_1}\ )\, 2\pi\xi_1).$$

Once a predefined number of photons have been scattered throughout the scene, the photons must be sorted in a manner in which they can be easily retrieved later. Jensen suggests using a k-d tree data structure to store the photons [Jen01].

Radiance Estimate

Storing the photons can be done in any data-structure, but during the second step of the photon mapping algorithm, the density needs to be found. The density of the photons is then used to estimate the irradiance. Density estimation is the process of finding the density at a specific point. As the photons are distributed in 3D the probability of finding any photons at a random 3D point is zero. Therefore techniques have to be used to give a measure of the density at a specific position. The most popular method is the N-nearest neighbors method. A fixed number of nearest neighbors are chosen and these nearest photons are found. The energy of all these photons is summed up and divided by the area that the photons span. By using this method the outgoing radiance can be described in the following way:

$$L_r(x, \varpi) = \int_\Omega f(x, \varpi', \varpi) \frac{d^2\phi_i(x, \varpi')}{dA_i} \approx \sum_{p=1}^{n} f(x, \varpi', \varpi) \frac{\phi_p(x, \varpi)}{\Delta A}$$

The area which these photons span can be calculated in different ways. One method is to use the convex hull of the photons. A faster but less accurate method is to make a sphere around the photons. Since speed is very important, as the density has to be calculated many times, the sphere method is often the method of choice. Finding the N-nearest photons can be quite time consuming. Therefore, it is desirable to store the photons in a data-structure that makes queries for the N-nearest neighbors easy. A kd-tree is fast to query for the N-nearest neighbors. Therefore the kd-tree is traditionally the preferred data-structure.

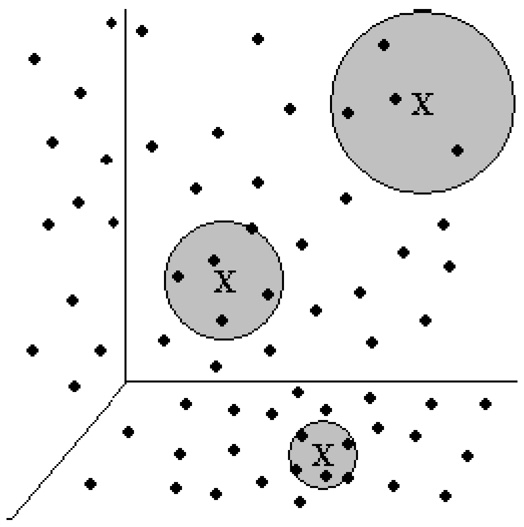


Figure 3: Radiance estimates are visualized [Joz02].

2.3.2 The Second Pass: Photon Rendering

The second pass of the algorithm creates an image in much the same fashion as ray tracing. It traces rays from the camera point through each of the pixels and finds the first surface the ray encounters. The color given to the pixel depends on the type of surface at the intersection point. The surface can by specular or diffuse and we use aforementioned equations to generate new ray or several rays in case of diffuse surface. In order to apply the irradiance (light to object) and the radiance (light to eye) to the point of interest, however, we must incorporate the photon map into the solution. For every reflected ray that hits another diffuse surface, a *radiance estimate* at the intersection point is taken. Once all of the sample radiance estimates are found, they are averaged together. This estimates the indirect illumination that the intersection point receives. This can then be combined with the direct illumination from the light sources and any specular properties to create an image using a global illumination model. This second pass is sometime called reconstruction, where photons are used to calculate indirect lighting and caustics. Direct illumination is calculated by using classical ray tracing.

Reconstructing the Caustics

The reconstruction of the caustics is done by using density estimates in the caustics photon map at the current sample point. Caustics are a high frequency effect and many photons are therefore needed to reconstruct the caustics accurately. In order to avoid blurred caustics it can be desirable to use filtering together with the density estimates ([Jen01]). The filtering process is a weighting of the photons where photons near the sample point is weighted higher than photons further away from the sample point.

Reconstructing the Indirect Illumination

The indirect illumination is calculated using a Monte Carlo based final gathering method. At the sample point (y), at which the indirect illumination is calculated, a vast number of rays are traced (usually 500-2000). At each hit location x, the outgoing radiance is found. The outgoing radiance can be calculated by using the irradiance and the BRDF of the surface. The irradiance is found as described above by using density estimation. The outgoing radiance from all these surfaces in the direction of the sample point is then used to find the incoming irradiance at the sample point. This incoming radiance is then used together with the BRDF of the surface of the sample point to find the radiance at the sample point (y). Thus the sample point is only illuminated by the light of all other surfaces and not by the light of the light sources. Therefore it is exactly the indirect light that is accounted for when one uses this method. If the indirect illumination is calculated directly by using the photons at the sample location as we do with the caustics, the image will be noisy. The reason for this difference in the reconstruction is that caustics are a high frequency effects while indirect illumination is a low frequency effect.

3. Approximating the Rendering Equation in Real-time

Main objective of this work is to simulate in real-time as many visual effect as possible that result from global illumination model. This is very active area this time because of growing power of new GPUs. Still not all effects is available in real-time, at least not simultaneously and certainly not with the same accuracy as if one of the methods for realistic image synthesis had been used. To solve integral relationship of global illumination we must either significantly simplify calculation, or otherwise came up with simple alternative that visually gives same results. Some of the methods presented in the literature for real-time approximation is described in this chapter. In this chapter the summarization of Pre-computed Radiance Transfer, Light Mapping and Real-time Photon Mapping Simulation comes from [Rev04]

3.1 Ambient Occlusion Techniques

In [Osi05] Osiris Pérez describes these techniques.
Ambient occlusion significantly differs from most global illumination techniques in the fact that it does not take light directly into account. Ambient occlusion just determines accessibility information of a surface based on nearby geometry. With this information we can modulate lighting intensity resulted from any other technique. We will use this technique witch conjunction with Phong shading model. Formally: The ambient occlusion A at a point P with a surface normal n is defined

$$A(P,n) = \frac{1}{\pi}\int_{\Omega} V(\omega,P)\max(\omega \cdot n, 0)\, d\omega$$

where represents ω various directions at P along the unit hemisphere Ω . V is visibility function returning 0 if no geometry is visible in the direction ω and 1 otherwise. In next sections we refer to the element that is shaded as the receiver an to the element that cast the shadow as the emitter.

3.1.1 Raytraced Methods

In this technique we determine accessibility and bent normal of every point of the surface with casting rays. Bent normal represents average direction of unoccluded incident light and is used in place of the regular normal when shading the surface for more accurate environment lighting. Accessibility can be easily computed as number of unoccluded rays divided by number of all rays. One way to generate rays is with rejection sampling: generating rays in the 3D cube. But better distribution can be ensured with Monte-Carlo sampling. The results are usually stored in texture and must be recomputed every frame for dynamic objects.

3.1.2 Depth map Methods

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

3.1.3 Dynamic ambient occlusion

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. Disk has its own position, normal and area. One disk is created per vertex. 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:

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

So when we are dealing with polygons then n = 3 and A_f is computed by using Heron's formula. 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. As approximation is used approximation based o the solid angle of an oriented disk to calculate the amount by which element shadows a receiver element. When A is the area of the emitter, amount of show is approximated:

$$1 - \frac{r cos\theta_E \max(1,4 cos\theta_R)}{\sqrt{\frac{A}{\pi} + r^2}}$$

All calculations are performed on GPU in the real-time, but big improvement comes from using hierarchical data structure. Because of soft shadows cast under diffuse lighting are generally soft, it is possible to approximate distant objects by simplified geometry. So we can group neighboring elements and represent them with larger element. Area of larger element is sum of its children’s and other attributes like positions, normals, etc. are averaged together. We traverse the hierarchy of elements and only if receiver is too close to emitter we need to traverse its children. This approach improve performance of algorithm from O(n^2) to O(logn).

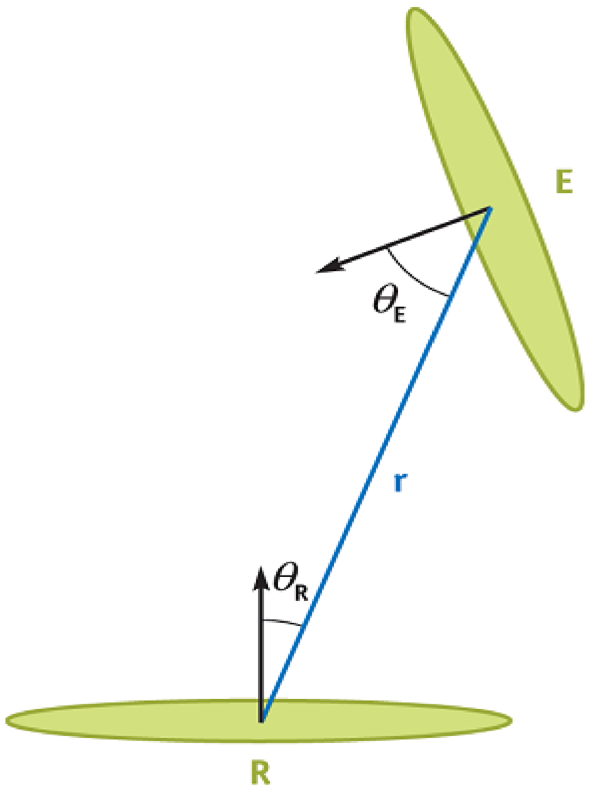


Figure 4: The Relationship Between receiver and Emitter Elements [Bun05]

3.1.4 Screen Space Ambient Occlusion

This approach originally comes from Crytek team [Mit07]. It is based on sampling surrounding of the pixel and with depth comparisons it is possible to compute occlusion factor. This method is limited to only nearby receivers.

We compute ambient occlusion in two passes. Design of the method is as follows:

1. Render scene normally and store z-buffer and normals in off-screen buffer.
2. Render full screen quad. Because we want to invoke fragment shader on all pixels.
3. In fragment shader we recover from 2D coordinates 3D coordinates in eye space.
4. Generate few points in a sphere around that eye position. These points are inside a sphere and on the surface of a sphere. According to the Crytek paper we use random reflection trick because of ugly banding artifacts.

5. Project these points to the screen space and next to the texture space.
6. From depth texture we have z coordinate of sampling points and fragment. We calculate difference between these points. Now we can calculate occlusion factor. Occlusion factor is based on this z-difference and we use quadratic attenuation function.
7. Because of noisy results it is suitable to blur the image. We use two-pass Gaussian filter.

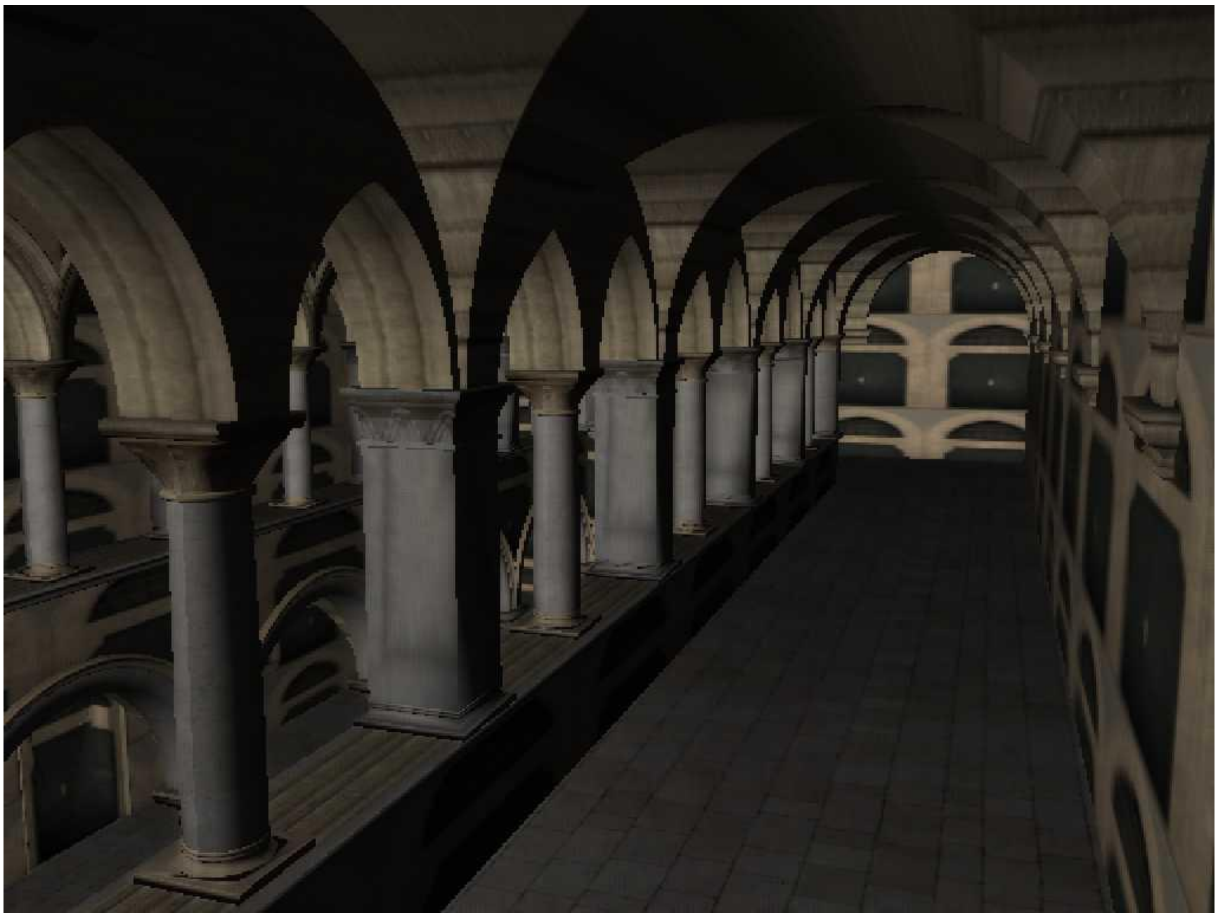


Figure 5. Sponza Atrium scene rendered with AO.

3.1.5 Hardware Accelerated Ambient Occlusion on GPU

We describe three methods according to [Sha07].

1. Image Space Approach For high-Frequency Ambient Occlusion

This method is based on approximating pixel by sphere. Idea is to perform ambient occlusion for those points that are visible to camera. So we only care about pixels that are in the Z-buffer. In fragment shader we can obtain the position P and Z-coordinate for given camera pixel. This couple is called ND-buffer (normals/depth). For each pixel on screen we compute following sum:

$$A(P,n) = \sum_{|P-Q_i|<r_{far}} A_\Psi(Q_{i,}r_i,P,n)$$

where

$$A_\Psi(C,r,P,n) = S_\Omega(P,C,r) * \max(n * PC, 0)$$

$$S_\Omega(P,C,r) = 2 * \pi * (1 - \cos(sin^{-1}(\frac{r}{|PC|})))$$

that means S_Ω is the surface area of the spherical cap subtended by the sphere <C, r> at unit hemisphere Ω. To control number of pixels gathered r_{far} is used. A_Ψ is ambient occlusion due to sphere $< Q_i, r_i >$. r_i , the radius of the sphere is function of the depth of the pixel q_i
We can use control the way looking-up pixels in ND-buffer. We can gather pixels in the square around given pixel or we can sum n random pixels in limited distance.

2. Distant-occluder Approach For Low-Frequency Ambient Occlusion

We can choose ε value as very small number. This number means smallest possible occlusion we care about. Let $d_{far} = |PC|$. From praefatus equations we can get

$$d_{far} = r * \frac{1}{\sin(cos^{-1}(1 - \frac{\varepsilon}{2\pi}))}$$

Now we project d_{far} onto image plane to obtain projected distance d'_{far}. When we cover all pixels in ND-buffer that are within a distance d'_{far} from C',we ensure that we cover all pixels that receiver at least ε ambient occlusion. A_Ψ is calculated for all pixels that are at most d'_{far} apart from C'.

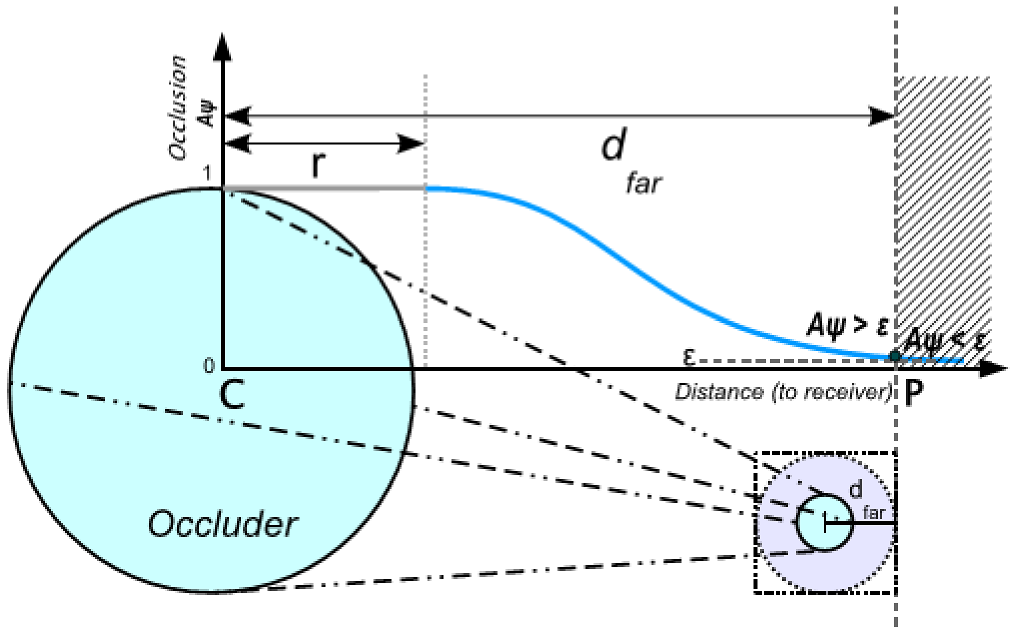


Figure 6. Parameter-ε explanation [Sha07]

3. Combined Ambient Occlusion

We obtain two buffers from using these two algorithms. Because of overlap between far and near occluders, we set

$$A_{\psi}(C,r,P,n) = 1 \; if \; P \; in < C,r >$$

in the case of the distant occluders approach. This combine these buffers additively and next we blend with color buffer by (Color*(1-Occlusion)).

One main drawback of this method is that over-occlusion artifacts may occur. This happens if multiple neighboring spheres contribute to the same pixel.

3.2 Real-time Caustics

For generating caustic we describe caustic mapping [Mus05] algorithm. This algorithm has simplistic nature of shadow mapping. So it supports fully dynamic geometry, lighting, viewing.

1. From light view the geometry is rendered to position texture.

2. Creating caustic map. In this step we refract light at each vertex, estimate intersection point of refracted ray within the receiver geometry, estimate caustic intensity at the intersection point. Caustic estimation is iterative process
3. Optional is construction of shadow map
4. From camera view we render the scene. We project each pixel into light's view to compute texture coordinates for indexing caustic map. Caustic map color is assigned to the pixel.

3.3 Light Mapping

One obvious advantage of radiosity compared to other global illumination techniques is that it computes lighting throughout the scene, and that this lighting is static as long as the objects do not move. The reason is the assumptions of diffuse surfaces only. Diffuse surfaces reflect light equally in all directions over the hemisphere and the resulting view-independence has the effect that the contribution of light to surface can be captured in a texture and attached to the surface. The concept of lighting information attached to a surface using a texture is referred to as light mapping. After the light maps have been created they can be used for real-time rendering nearly without expenses. In diffuse environment it will even be difficult to notice that the lighting does not change accordingly when objects are moved around. Still light mapping is an inexpensive and widely used technique which can simulate diffuse reflections in real-time. In [Bri04] it is described how radiosity was calculated on graphics hardware and stored in light maps for real-time use in a large scale from Atari called "Shadow Ops".

Other global illumination techniques could be used with light mapping as well as radiosity can. We must always make sure, though, that only the multiple diffuse reflections term is computed, if we plan to combine the light mapping with other methods. The reason why it is clever to use light maps only for multiple diffuse reflections term is that the indirect illumination changes quite slowly over the surfaces and therefore it will be less noticeable that such lighting is static.

To summarize light mapping can create sophisticated lighting at minimum cost for static scenes illuminated with static light sources. An optimization of the concept is to store only light reflected diffusely at least once in the light map, and then combine the light mapping technique with other methods for different parts of rendering equation. If we need our application to simulate dynamic indirect lighting, or we do not have time for a global illumination preprocessing stage, we have to look for different approaches to an evaluation of the multiple diffuse reflection term.

3.4 Pre-computed Radiance Transfer

In radiosity form factors represent energy transfer between surfaces. It is assumed that all surfaces reflect light diffusely and that radiosity is constant across the surface of a patch, hence, we need not consider the dependence on direction and location. This is why form factors represent energy transfer. If we remove the assumption of diffuse surfaces only, the concept of form factors can be generalized to represent irradiance transfer (instead of energy transfer) between surfaces. In practice this means that an arbitrary BRDF must included in the radiosity form factors. In [Fra91] it is shown how a collection of spherical harmonics basis functions can represent such generalized form factors, or irradiance transfer functions. The reason for using spherical is that “a finite number of terms can be used to approximate relatively smooth functions defined on the sphere [Fra91]

The concept of pre-computed radiance transfer, described in [Pet02], is to let a dense set of vectors or matrices over a surface represent its radiance transfer function. The radiance transfer function is a further generalization of the spherical harmonics based approach described in [Fra91]. The radiance transfer functions are sufficiently general to be used for the evaluation of the rendering equation directly.

Though arbitrarily complex BRDFs can be simulated using radiance transfer functions. The pre-computed radiance transfer methods often limit themselves to low-frequency lighting only. The reason is that diffuse surfaces and glossy surfaces with diffuse characteristics, typically require fewer spherical harmonics basis functions in the radiance estimate than specular surfaces do. Fewer basis functions means fewer computations and, hence, the method becomes less expensive and less prone to aliasing artifacts resulting from an insufficient order of the spherical harmonics basis.

The approach is then to pre-compute for a given surface the radiance transfer functions, which are independent of the incident radiance term. A summation over inner products between the transfer functions and incident radiance samples on a surface the gives an inexpensive, dynamic, approximate evaluation of the rendering equation.

Because of extensive pre-computations the method is restricted to use of rigid objects only. In [Pet02] a method is also described for neighborhood-transfer, where radiance transfer functions are also computed from a rigid body to its neighbor space. This expansion allows soft shadows, glossy reflections, and caustics on dynamic receivers.

The neighborhood-transfer important if the method is to be useful in a complex scene composed of many moving objects. However, neighborhood-transfer has problems when multiple objects reflect light or cast shadows on the same receiver. Moreover the lighting must by fairly constant across the entire neighborhood of an object to provide accurate results. This indicates that pre-computed radiance transfer is brilliant method if only a single objects are present in the scene at the same time. In more dynamic environments the method unfortunately falls apart. And lastly the developer using the method must be prepared for long pre-computation times.

3.5 Environment Map Rendering

Greger et al. [Gre98] used uniform grid for storage and interpolation of global illumination. The data structure and interpolation strategy used in this algorithm [Nij04, Man02] is similar. Process of algorithm is as following:

1. Divide 3D space into uniform grid of volumes.
2. Render a cube map at the center of each volume. The cube map gathers information about incoming radiance.
3. Compute spherical harmonic coefficients for every cube map. These coefficients represent the vector irradiance at the center point of the volume. Irradiance for a Lambertian surface point can be computed from coefficients and the normal to the surface at that point.
4. Compute irradiance at each surface vertex by linearly interpolating irradiance of 8 neighboring volumes. Irradiance is assumed to vary smoothly both with direction and position. Use the irradiance value to set the ambient color for the vertex.

Irradiance volumes are described as extended versions of irradiance, which is defined in all points and directions in space. Idea is to combine the environment mapping representation of

transfer functions with the irradiance volumes approach to global illumination. Instead of sphere maps, cube environment maps are used for storage of radiance transfer functions.
Cubemaps are placed in regular grid 4x4x4 across the scene. Each environment map is updated each frame. According to Nijasure et al. there are only few limitations to their technique, and they propose solutions for all problems they describe. One problem is that the 4x4x4 grid dimensions is not sufficient for larger scenes. The solutions the propose is dynamic grid structure, where grid is laid out in the viewing frustum volume and the grid moves dynamically with the view.
In their implementation they have restricted to radially symmetric BRDFs but they noted their algorithm is equally applicable to scenes with arbitrary non-diffuse surfaces.
The reported results are still quite impressive.

3.6 Real-time Photon Mapping Simulation

Real-time implementation of photon mapping is presented [Lar04, Chr4]. The real-time simulation of photon mapping first seeks out the different surfaces in a scene. According to the method described in [Ben03], a global map is created for each surface, which speeds up photon map construction and radiance estimated.

A variant of the selective photon emission described in [Dmi02], is used to emit only intelligently selected photons each frame. Using selective photon emission, the photons are emitted in groups. The method presented in [Chr04] emits one photon for each frame for each group. The path of the previous photon emitted in each group is stored. If the path taken by a photon in a group is not the same as the previous path, the entire group of photons is marked for redistribution.

For each surface a texture is generated holding radiance estimates from the photon map related to that particular surface. These textures are referred to as approximated illumination maps (AIM).

A hardware optimized final gathering method is presented, where number of locations on the surfaces of the scene is chosen. AIMs are applied to each surface and then a picture is taken from each chosen location to simulate final gathering rays. The resulting pictures are averaged using hardware optimized mipmapping. The result is a number of pre-determined locations distributed over the surfaces throughout the scene, where the multiple diffuse reflections terms is known – we could call it an indirect illumination field.

One way to visualize indirect illumination field is simply to store the indirect illumination in coarse textures, which can then be applied to the surfaces, where the indirect illumination was calculated. This approach is feasible since diffusely reflected indirect illumination usually changes slowly over a surface.

Compared to light mapping this method handles dynamic scenes illuminated by dynamic light sources. Furthermore it potentially simulates all kinds of lights paths in the multiple diffuse reflection term (L(S\D)*DS*DS*E). The approximate final gathering unfortunately smooth out the result, but this is relative since it depends on the number locations we choose on the different surfaces. The resulting illumination in a Cornell box using this method is almost indistinguishable from the global illumination reference. Dynamic objects should be pointed out in advance and each of them should have an individual data structure.

This approach is not easily scaled to large and complex scenes, the number of approximated illumination maps that must be processed must will escalate.

3.7 Reflective Shadow Maps

Reflective shadow maps [Dac05] enable an approximation to first-bounce indirect illumination. The first-bounce lights can be found by looking at the shadowmap depth. By sampling all points of the shadow maps generated in a scene, it is easy to capture all secondary light sources. The result gives moderate color bleeding based on the locality of objects in the scene. RSM stores from light view positions, normals and radiant flux. Pixels of a shadow map are also considered as point lights. World space position can be recomputed from pixel coordinates or stored in RSM. In OpenGL it is possible to capture this information by using FBO(Frame Buffer Object). This information generates the irradiance defined as:

$$E_p(x,n) = \phi_p \frac{\max\{0, n_p * (x - x_p)\}\max\{0, n_p * (x_p - x)\}}{\left\|x - x_p\right\|^4}$$

The summed irradiance from all pixels the shadow map represents the total contribution from indirect lighting in the scene. Because of full solution is not feasible we can choose point lights according to a uniform sampling rate. According to the paper we project fragment into shadow map and sample radially outwards with uniform random variables over a sphere of some predefined radius. This approach does not consider occlusion for indirect light sources so some points are illuminated incorrectly. Despite of this it can lead to very wrong results but in most cases it suffices. It is proposed to apply Ambient Occlusion because of this method doesn’t handle self-shadowing for the indirect light.

3.8 Summary

Technique	Advantages	Disavantages
Light Mapping	supported through multitexturing, fast	static geometry, static lighting
PRT	dynamic lighting, extension for glossy surfaces, multi-bounce	static geometry, preporcessing, distant lighting
Environment Map Rendering	impressive results, good scalability	dependent on number of polygons, slow
Real-time Photon Mapping Simulation	gpu implementation exists	realtime for small scenes, dynamic objects should be pointed out, noisy
Reflective Shadow Maps	one bounce, dynamic geometry and lighting	no occlusion for indirect light sources
SSAO & HAAO	self-shadowing, lighting independent	only nearby receivers, only for objects in zbuffer

We see that for dynamic complex scenes and deformable geometry with dynamic light sources are most suitable SSAO and HAAO. Therefore we implemented these techniques and results are in chapter 4.

4. Implementation

Some of proposed methods were verified by concrete implementation. This will be described in this chapter.

4.1 Introductory information

4.1.1 Purpose of Program

Application was developed in order to prove proposed algorithms. Purpose of the program is to calculate global illumination from input geometry with respect to real-time i.e. <=65ms per frame (15fps).

4.1.2 Operating Description

Program loads description of the scene and configuration data from disk. Rendering core begins compute lighting for visible parts of the scene. Parameters of calculation can be easily changed by settings in menu or key shortcuts. Programs renders virtual environment, in which user can free walk and also move with objects or with virtual light sources. Rendering is provided by graphic APIs OpenGL or Direct3D. Parameters of camera can be controlled by keyboard and mouse.

These main functions are available:

- Reads scene from 3DS, OBJ, MD2
- Navigation in 3D environment
- Moving with objects and lights
- Calculating global illumination with maximal effectiveness
- Utilize 3D graphic accelerator to render
- Allow store current frame to file
- Installer program

4.1.3 Input and Output

Input scene can be stored in 3DS or OBJ. Both formats describe scene by boundary representation and contain basic material properties of surfaces. Images can be stored in BMP, JPG and TGA format. Output is real-time walkthrough through scene.

4.2 Program Structure

4.2.1 Development Environment

Program Real-time Global Illumination is object oriented application. C++ language is used for implementation. We chose C++ because it is object-oriented language suitable for building 3D Engine. C++ is excellent in speed, where other languages for example Java are behind.

To create user interface it is used development environment Microsoft Visual C++ 2005. Main data structures are platform independent.

Program is intended for Windows XP Operating System. Part of rendering core of our application uses OpenGL 2.1 with support FBO and floating point textures. GLEW library is used for handling OpenGL extensions. We used jpeg lib for manipulating with images. For developing shaders we chose TyphoonLabs ShaderDesigner. Shaders are developed according to OpenGL shader language specification 1.2.

4.2.2 Model Design

In first step we analyzed the problem and we created detailed design of program structure.

Application is divided into windows-system specific parts and platform independent parts.

As window-system specific parts we mean user interface. Figure 7. Displays UML class diagram and represents program structure and relationship between classes.

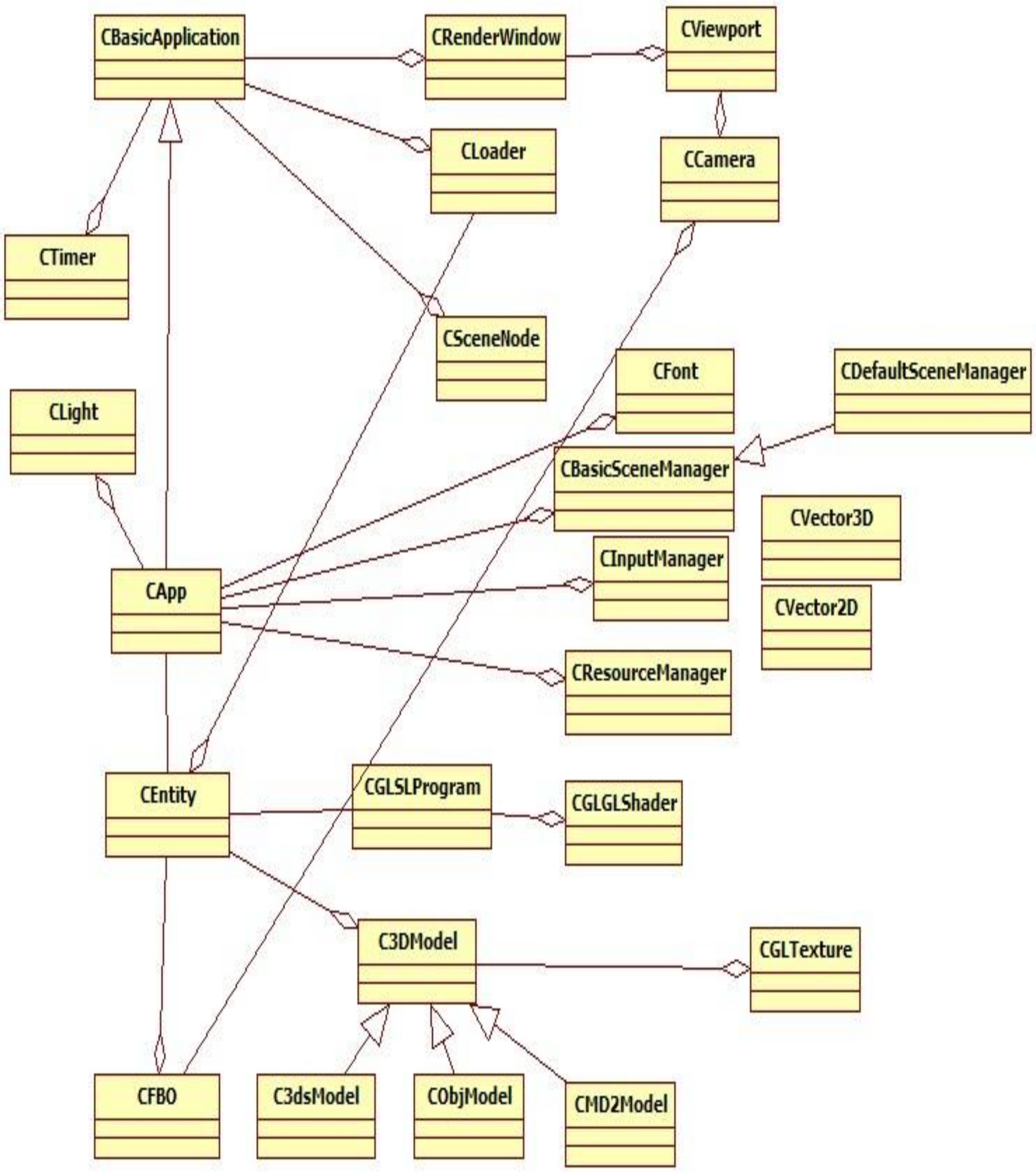


Figure 7. UML class diagram

<u>CBasicApplication</u>

Main class. Provides basic interface between application specific data and rest of application. Also connects OpenGL rendering content to window device content.

CApp

As descendant of CBasicApplication serves as basic input point to application. In this class it is possible by overriding one method define the scene, control the rendering and loading data. For this purpose it can use some helper classes and managers.

CRenderWindow

This class is used by CBasicApplication and can be used to read and to set window attributes.

CViewport

Stores viewport attributes as width, height, minimal x-coordinate and minimal y-coordinate and camera.

CCamera

This class takes care about transformations virtual camera controlled by user interface system and its state variables are updated each frame.

CLoader

This class is used for loading data to memory. For this purpose is created new thread which dealing with keeping actual list of data to proceed.

CTimer

Moving camera and swiftness of application is based on actual frame rate. This class computes frame rate and can limit speed of rendering scene to defined fps(frames per second).

CSceneNode

This class represents node in scene graph. That means CSceneNode keeps number of its childrens, transformations relative to his parents and entities needed to render.

CFont

can put characters to the screen. Freetype library is used.

CBasicSceneManager

is basic abstract class used by its descendants. It contains methods for controlling the rendering, displaying additional informations as normals, lights. It contains info about number of passes needed. Class takes care about pass info of each entity and let only those that are set for actual pass.

CDefaultSceneManager

is default implementation of CBasicSceneManager class. This class does not provide methods for culling polygons as Frustum Culling or Octal Tree. Its purpose is only render all data that it gets. In OpenGL implementation is used vertex buffer objects or vertex arrays if not available.

CInputManager

hides functions needed for handling mouse and keyboard. Implementation is done only for Windows Operating System.

CResourceManager

determines absolute locations of each resource type(textures, models, shaders, fonts) from resource file and provides this information to correspondent classes.

CEntity

carries information about render mode (Perspective, Ortho), pass, shader, 3D model, and FBO(Frame buffer object).

CGLSLShader

Internal class. This class is used for loading, compiling and deleting shaders (Vertex shader, Fragment shader) from text files.

CGLSLProgram

This class uses CGLSLShader class and provides communication between shaders and application. To work it use CGLSLShader class.

CGLTexture

carries information about texture. This class is OpenGL specific. It enables loading image form various formats and bind them as OpenGL texture. Currently supported formats are bmp, tga, jpg.

CFBO

FBO represents off-screen buffer in OpenGL. This class merges all attributes needed to use it. These attributes involves frame buffer object, render buffer, camera and textures associated with this FBO.

C3DModel

is abstract class for C3dsMOdel, CObjModel and CMd2Model. It stores all material data (ambient, diffuse, specular, shininess), vertex data (position, normal, texture coordinate), mesh data (indices, name, material ID) and texture data needed to render given model.
To methods of this class belongs methods for computing normals for given faces and convert data to representation suitable for rendering with vertex arrays.

C3dsModel

Implementation of C3DModel. It enables loading of 3DS files. Current implementation loads only vertex data, mesh data, materials, and texture informations.

CObjModel

Implementation of C3DModel. It enables loading of OBJ files. It supports loading materials from additional MTL file.

CMd2Model

Implementation of C3DModel. It enables loading of MD2 files. MD2 format is format used by Quake2 game and contains key frame animation data. Implementation can read this data and is able to do transformations of vertices on vertex shader.

4.3 Experiments and Verification

We perform experiments on an AMD Athlon64 X2 4400+ (2,3GHz), 2GB RAM, graphic card GeForce 8800GT 512MB DDR3. As test scene we used Sponza Atrium and Sibenik stored in 3DS format. These models are considered as reasonably big scenes and they allow us to compare results of proposed methods with other rendering methods. We provide various settings of parameters and we record speed by fps counter and check visual quality by human eye. Implementation SSAO and HAAO was implemented according to chapter 3.1.4 and 3.1.5. Test scenes Sponza Atrium and Sibenik were developed by Marko Dabrovic.

1. Screen Space Ambient Occlusion

This method is based on sampling fragment around sphere a computing occlusion factor. Results are dependent on number of samples, sphere radius, occlusion factor computation. Our sphere has radius 15. Occlusion factor is computed only if sampled z-coordinate is smaller as fragment z-coordinate. We try different scenes and number of samples.
For Screen Space Ambient Occlusion fragment shader we use following piece of code:

```
  ...
  float occlusion = 0.0;
  vec4 se;
  vec4 ss;
  vec4 sn;
  float zd;



  float sampleDepth;
  for( int i = 0; i < numSamples; i++ )
 {
    vec3 curSample = samples[i].xyz;
    curSample = reflect(curSample, randNormal.xyz);
    if( dot( curSample, origNorm.xyz) < 0.0 )
    {
                    curSample = reflect(curSample, origNorm.xyz);
    };

    se = vec4( eyePos + ( radius * curSample ), 1.0 );
    sn = se * gl_ProjectionMatrix;
    ss.xyz = sn.xyz / sn.w;
    ss.xy = ( ss.xy * vec2( 0.5, -0.5 ) ) + vec2( 0.5, 0.5 );

    float sampleDepth = texture2D(positionTexture, ss.xy).w * (1.0);
```

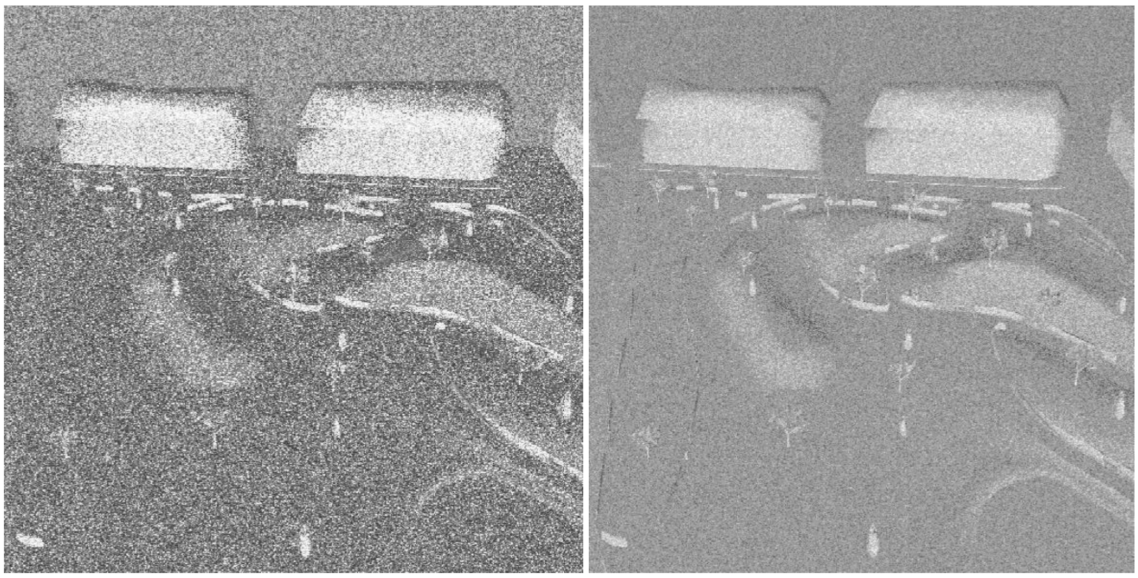
```
        vec4 sampleColor = texture2D(colorTexture, ss.xy);
        zd = ( se.z - sampleDepth );

        if ( se.z < sampleDepth )
       {
                        float falloff = 1.0 / ( 1.0 + ( zd * zd * 0.01 ) );
                        occlusion += falloff;
       }

   };
  gl_FragData[0] = vec4(occlusion/float(numSamples));
  ...
```

Results:

1. Test Scene: Park, 29174 polygons.



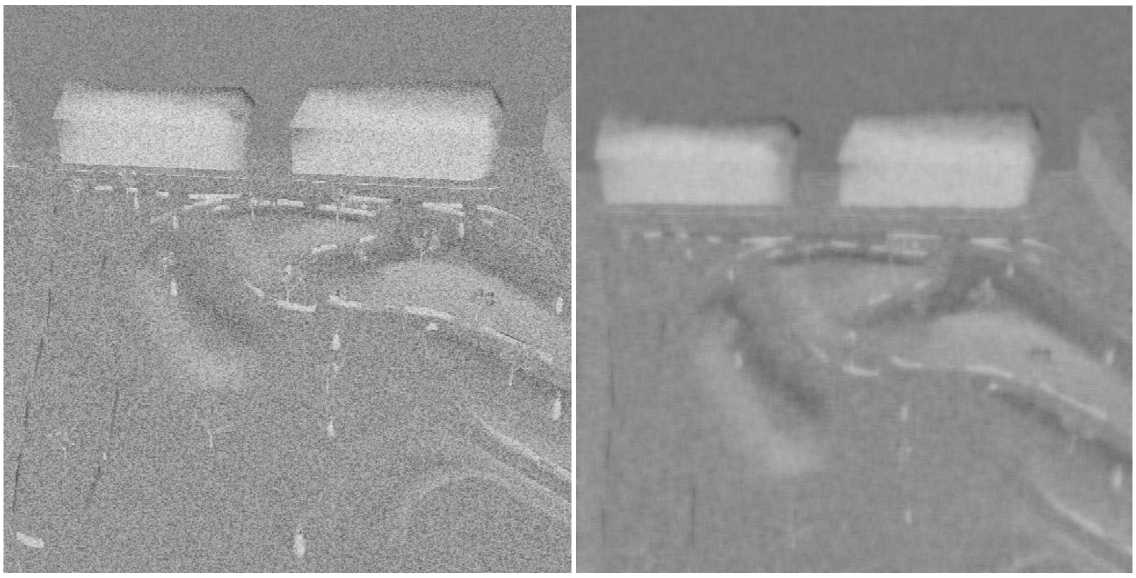
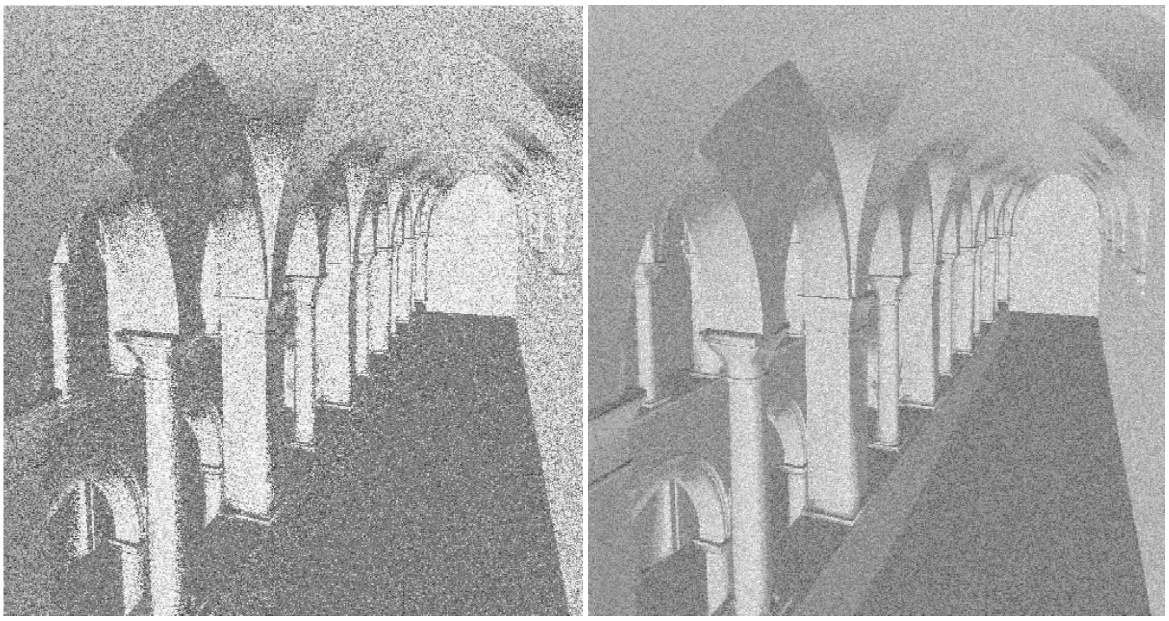


Figure 8. Park with SSAO. Number of samples from left to right, top down: 8/16/32/ blurred 32

2. Test Scene: Sponza Atrium, 66454 polygons.



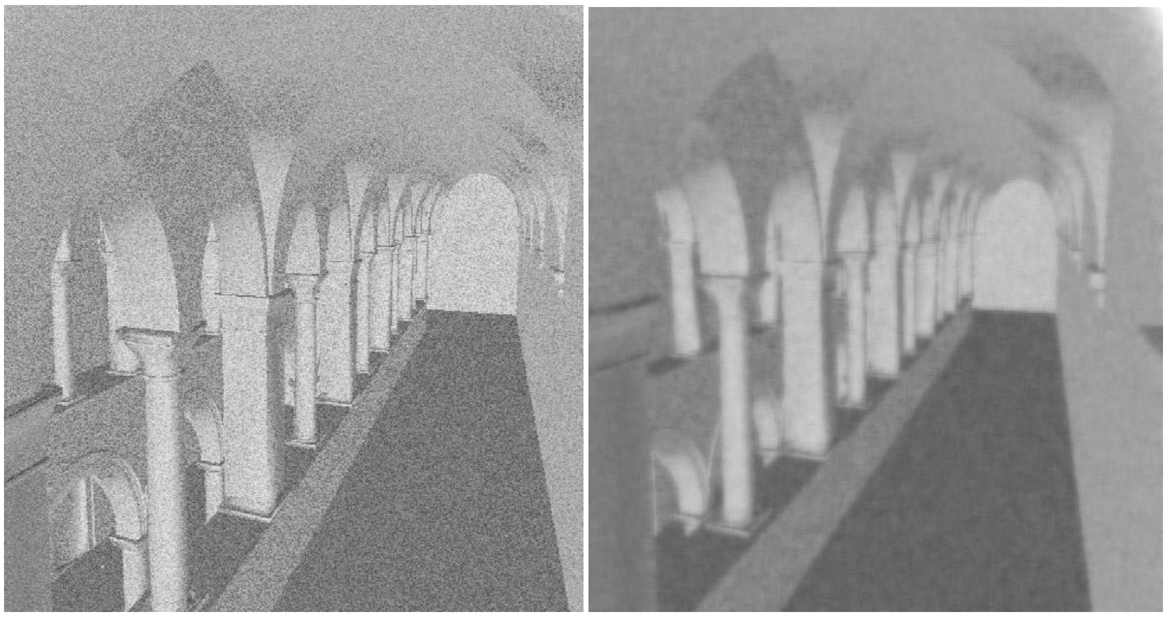
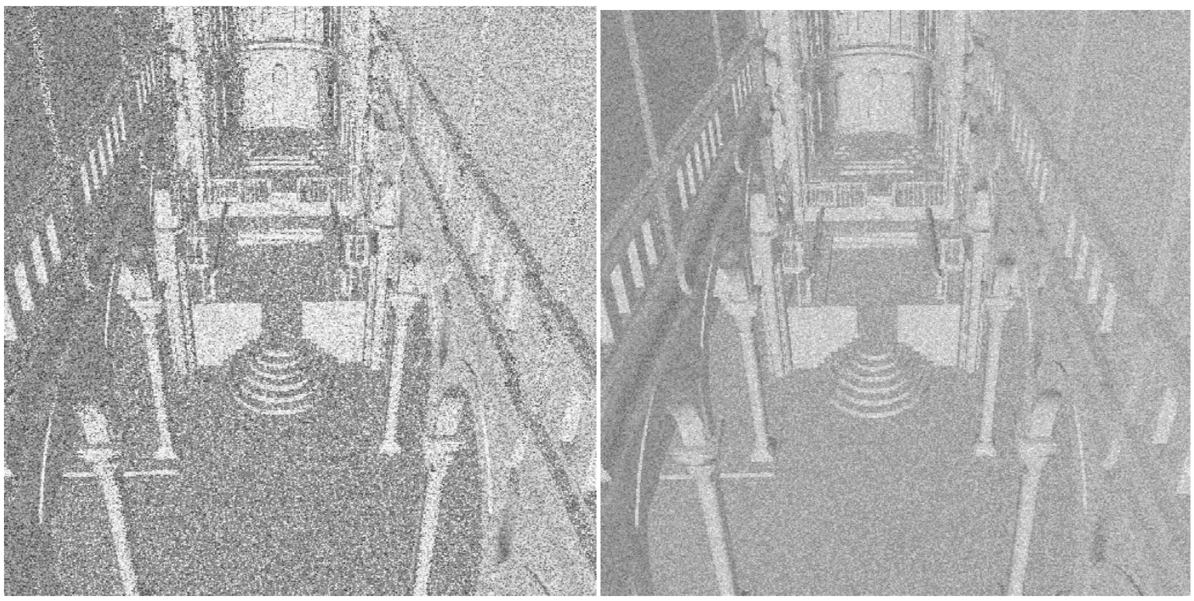


Figure 9. Sponza Atrium with SSAO. Number of samples from left to right, top down: 8/16/32/ blurred 32

3. Test Scene: Sibenik, 80479 polygons.



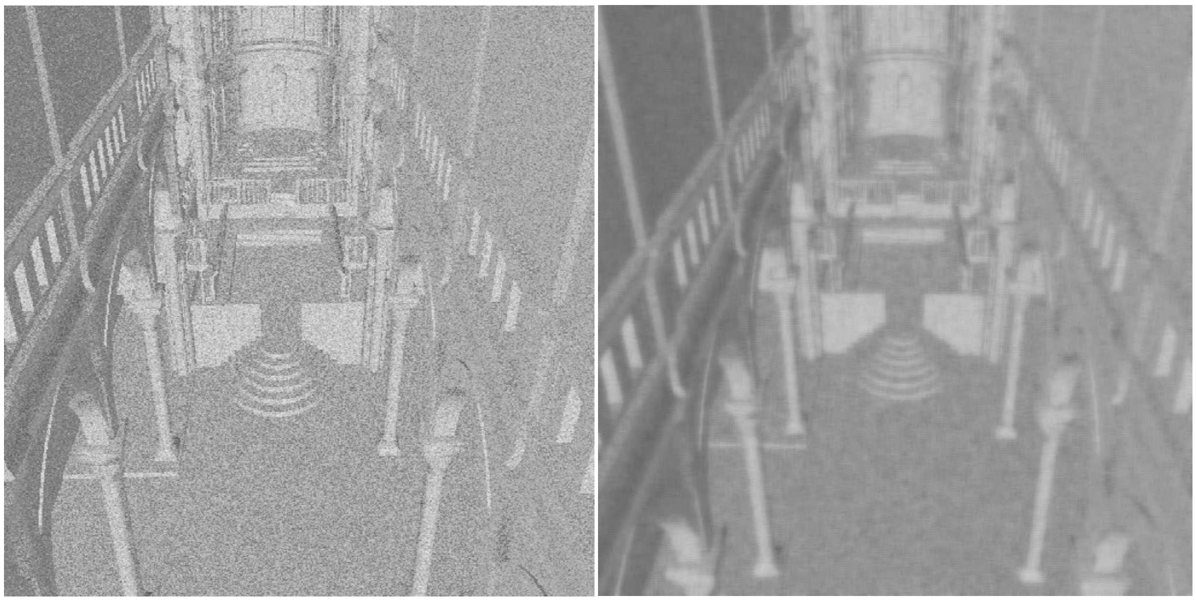


Figure 10. Sibenik with SSAO. Number of samples from left to right, top down: 8/16/32/ blurred 32

Scene/ Samples	8	16	32	blurred 32
Park	39	38	25	20
Sponza Atrium	38	38	21	18
Sibenik	38	31	24	17

Figure 11. FPS for various scenes and samples

2. Hardware accelerated ambient occlusion

We implemented only image space approach according to [Sha07]. As sampling scheme we used sampling a simple square area around given pixel.
For Hardware Ambient Occlusion fragment shader we use following code:

```
float SurfArea(vec3 ReceiverPos, vec3 SamplePos, float Radius)
{
        float arg = asin(Radius/distance(ReceiverPos, SamplePos));
        return 2*PI*(1-cos(arg));
}
```

```
float Afi( vec3 SamplePos, float Radius, vec3 ReceiverPos, vec3 ReceiverNormal )
{
        float param = dot(ReceiverNormal,(SamplePos-ReceiverPos));
        return SurfArea(ReceiverPos, SamplePos, Radius)* max(param, 0.0);
}



void main()
{
        vec2 vTexCoords[9];
        float fTexelSize = 1.0 / 512.0;

        vTexCoords[0]= gl_TexCoord[0].st;
        vTexCoords[1]= gl_TexCoord[0].st +  vec2( -fTexelSize, 0.0 );
        vTexCoords[2]= gl_TexCoord[0].st +  vec2(  fTexelSize, 0.0 );
        vTexCoords[3]= gl_TexCoord[0].st +  vec2( 0.0, -fTexelSize );
        vTexCoords[4]= gl_TexCoord[0].st +  vec2( 0.0,  fTexelSize );
        vTexCoords[5]= gl_TexCoord[0].st +  vec2( -fTexelSize, -fTexelSize );
        vTexCoords[6]= gl_TexCoord[0].st +  vec2(  fTexelSize, -fTexelSize );
        vTexCoords[7]= gl_TexCoord[0].st +  vec2( -fTexelSize,  fTexelSize );
        vTexCoords[8]= gl_TexCoord[0].st +  vec2(  fTexelSize,  fTexelSize );

        float depth = texture2D(positionTexture, gl_TexCoord[0].st).w;
        vec3 ReceiverPos = normalize(eyeRay) * depth;
        vec3 ReceiverNormal = ( texture2D( normalTexture, gl_TexCoord[0].st).xyz * 2.0 ) –
         vec3(1.0);

        float occlusion = 0.0f;

        for( int i = 1; i < 9; i++ )
        {
                vec3 SamplePos = texture2D(positionTexture, vTexCoords[i] ).xyz;
                float depth = texture2D(positionTexture, vTexCoords[i]).w;
```

```
            occlusion+=Afi(SamplePos, fRadius, ReceiverPos, ReceiverNormal);
        };

        gl_FragData[0] = vec4(occlusion);
}
```

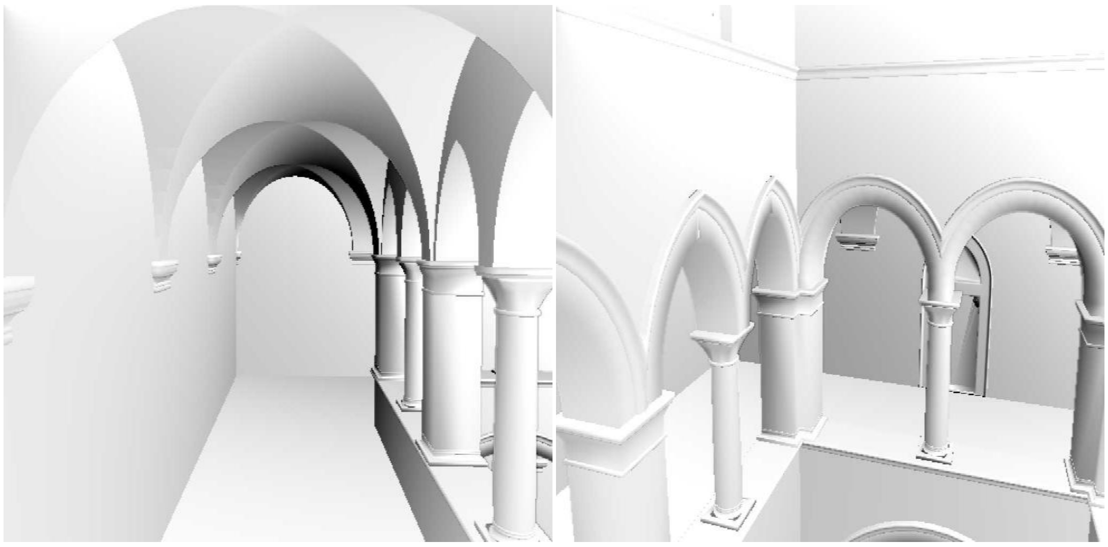


Figure 12. Sponza Atrium with HAAO

4.3.1 History of Development

In first step we analyze each algorithm and determine its advantages and disadvantages.
A lot of techniques have limitation only to static lighting or static geometry. Next limitation is dependency on number of polygons. Therefore our choice was aimed on screen space methods. These methods contain methods based on shadow maps and screen space ambient occlusion techniques. Major computations are done on fragment shader.
Subsequently we design UML diagram for simple manipulating with shaders.
We developed program that demonstrate results from screen space ambient occlusion and hardware ambient occlusion techniques. Proposed methods are suitable for fully dynamic scenes and are lighting independent.
Encountered problems:

1. Problems with shader debugging. Shader programming is still relatively new discipline. Most tools are able to develop sophisticated shaders but shader debugging is missing. We aim only on GLSL shaders and use tool called GLSL devil for shader debugging. Unfortunately on our configuration, output of all shader variables was disabled.
2. We noticed that dynamic indexing of arrays in fragment shader is not supported on Geforce 7 series GPU. This small problem is eliminated on Geforce 8 card or by setting numSamples in SSAO fragment shader as constant.

4.3.2 Current State

These methods proceeded through a lot of testing and optimizing. User interface was implemented in the form of window application. Communicating with program is possible through mouse and keyboard. Parameters can be changed with hot keys.

4.3.3 Future Work

A lot of improvements are possible. Shader code can be improved in many ways. For example in SSAO implementation can be tuned radius of the sphere, number of sampling points, distribution of the points, blocker attenuation function, blur kernel, contrast adjustments…

5. Conclusion

Diploma thesis provides systematic overview of global illumination methods. Global illumination methods are divided into classic methods and real-time approximations. Results of diploma thesis can be summarized into following points:

- Diploma thesis aims on exploration of global illumination methods. We provide overview of classic algorithms.
- Main attention was dedicated to real time approximations of rendering equation. We presented wide range of real-time solutions. We analyzed solution from perspective of dynamic scenes, dynamic lights and deformable geometry. All of proposed solutions are limited to diffuse surfaces.
- Implementation deals with Ambient Occlusion Lighting. First implemented method, SSAO is based on computing distant dependent occlusion factor for each pixel. Second method, HAAO is dealing with approximating pixel by sphere and computing spherical cap for each pixel.

6. Glossary

BRDF

The relationship between incoming light (E_i) and outgoing light (L_r) at a point (x) can be described using a function of three parameters. These parameters are the point (x), the incoming direction of light ($\vec{w}'$) and the outgoing direction of light ($\vec{w}$): $f(x, \vec{w}, \vec{w}') = \frac{dL_r(x, \vec{w})}{dE_i(x, \vec{w})}$, where f is called Bidirectional Reflectance Distribution Function.

Radiance

radiometric measures that describe energy that leaves a surface, per unit projected area of the surface per unit solid angle of direction. Mathematically this can be expressed as:

$$L = \frac{d^2\phi}{dAcos\theta d\varpi}$$

Irradiance

Irradiance E is total power per unit are incident onto a surface with fixed orientation

Reflectance

Reflectance(ρ) is the relative amount of the incident light that is reflected. At a surface point (x) this can be expressed as:

$$\rho(x) = \frac{d\phi_r}{d\phi_i}$$

Caustic

is the envelope of light rays reflected or refracted by a curved surface or object, or the projection of that envelope of rays on another surface

Radiance transfer function

maps incoming radiance to outgoing radiance. Complex effects like interreflections or caustics are represented in the transfer function. Transfer functions are represented as dense set of vectors or matrices over its surface.

7. Bibliography

[Joz02] Jozwowski, Timothy R., Real Time Photon Mapping, Master Thesis, Michigan Techological University, May 23, 2002

[Pro03] Proňková, Lenka, Monte Carlo radiační metody, Matematicko-fyzikální fakulta, 2003

[Mar02] Marlon John, Focus On Photon Mapping, Premier Press, 2002

[Žár04] Žára Jiří and others, Moderní počítačová grafika, Computer Press, 2004

[Rev04] Jeppe Revall Frisvad and Rasmus Revall Frisvad, Real-Time Simulation of Global Illumination Using Direct Radiance Mapping, November 2, 2004

[Gla89] Glassner Andrew S.Editor, An Introduction to RayTracing, Academic Press, London, 1989

[Jen01] Jensen Henrik Wan, Realistic Image Synthesis Using Photon Mapping, AK Peters Natick, Massachusetts, 2001

[Shi00] Shirley Peter. Realistic Ray Tracing, AK Peters, Natick, Masachusetts, 2000

[Dmi02] Kirill Dmitriev, Stefan Brabec, Karl Mzsykowski and Hans-Peter Seidel. Interactive global illumination using selective photon tracing. In rendering technique ’02 (Proc. Of the Thirteenth Eurographics Workshop on Rendering), pages 21-34,2002)

[Lar04] Bent Dalgaard Larsen. Global Illumination for Real-Time Applications by Simulating Photon Mapping. PhD thesis, Informatics and Mathematical Modeling, Technical University of Denmark, 2004

[Ben03] Bent Dalgaard Larsen and Niels Jorgen Chrisensen. Optimizing photon mapping using multiple photon maps for irradiance estimates. In WSSCG’2003 Poster Proceedings, pages 77-88, February 2003

[Chr04] Bent Dalgaard Larsen and Niels Jorgen Christensen. Simulation photon mapping for real-time application. In H.W. Jensen and A.Keller, editors, Eurographics Symposium on rendering, 2004

[Bri04] Brain Ramage. Fast radiosity using pixel shaders. Game Developer,11(7):20-39, August 2004

[Fra91] Francois X.Sillicion, James R.Arvo, Stephen H Westing, and Donald P. Greenberg. A global illumination for general reflectance distributions. Computer Graphics (SIGGRAPH '91 Proceeding), 25(4):187-196, August 1991

[Pet02] Peter- Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. ACM Transactions on Graphics, 21(3):527-536, July 2002

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

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

[Mit07] Martin Mittring: Finding next gen: CryEngine 2, 2007

[Sha07] Shanmugam P., Arikan O., Hardware Accelerated Ambient Occlusion Techniques on GPUs, 2007

[Nij04] Mangesh Nijasure, Sumanta Pattanaik, Real-Time Global Illumination on GPU, 2004

[Man02] Rafal Mantiuk, Sumanta Pattanaik, Karol Myszkowski, Cube-Map Data Structure for Interactive Illumination in Dynamic Diffuse Environments, September 2002

[Dac05] Carsten Dachsbacher, Reflective Shadow Maps, 2005

[Osi05] Osiris Pérez, Real-Time Dynamic Ambient Occlusion, September 2005

[Gre98] Greger, G., Shirley, P., Hubbard, P. and Greenberg, D. P. 1998. The Irradiance Volume.IEEE Computer Graphics & Applications, 18(2), 32 – 43

[Mus05] Musawir Shah, Sumanta Pattanaik, Caustic Mapping: An Image-space Technique for Real-time Caustics, 2005