Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Graphics

Diploma Thesis

Voxel Cone Tracing for Indirect Illumination

Author:	Bc. Tomáš Dřínovský
Supervisor:	Ing. Tomáš Barák
Study Programme:	Open Informatics
Field of Study:	Computer Graphics and Interaction

May 8, 2013

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Tomáš Dřínovský**

Study programme: Open Informatics
Specialisation: Computer Graphics and Interaction

Title of Diploma Thesis: **Voxel Cone Tracing for Indirect Illumination**

Guidelines:

Review current state of the art real time algorithms for computing indirect illumination in dynamic scenes.
Implement the Interactive Indirect Illumination using Voxel Cone Tracing [1] method in OpenGL/GLSL and C++.
Test the implementation on five or more scenes.
Compare the results with other provided implementations, focus on rendering quality of glossy surfaces.

Bibliography/Sources:

[1] Crassin, Interactive Indirect Illumination using Voxel Cone Tracing, CG Forum, 2011
[2] Kaplanyan, Cascaded Light Propagation Volumes for Real Time Indirect Illumination, I3D, 2010
[3] McGuire, Hardware-Accelerated Global Illumination by Image Space Photon Mapping, Siggraph 2009
[4] Ritschel, Making Imperfect Shadow Maps View-Adaptive: High-Quality Global Illumination in Large Dynamic Scenes, EGSR 2011
[5] Cozzi, OpenGL Insights, 978-1439893760, CRC Press 2012

Diploma Thesis Supervisor: Ing. Tomáš Barák

Valid until the end of the summer semester of academic year 2013/2014

[illegible]

prof. Ing. Jiří Žára, CSc.
Head of Department

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-4-
L.S.

[illegible]

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 14, 2013

Acknowledgements

I would like to thank to the supervisor of this work Ing. Tomáš Barák for his insight and numerous helpful advices. Another thanks belong to my parents and my brother for their support during whole my studies and to my girlfriend for keeping me sane.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used. I have no objection to usage of this work in compliance with the act §60 Zákon c. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 8, 2013 Tomáš Dřínovský

....

Abstract

Indirect illumination is the fundamental part of realistic image synthesis. Its evaluation is computationally expansive and usually highly dependent on geometrical complexity of the scene.

This thesis deals with the real-time indirect illumination. We examine the state of the art algorithms in this field and one of them, the Voxel Cone Tracing introduced by Crassin et al. in 2011, is implemented. The implementation uses capabilities of a modern GPU and is capable to render diffuse indirect illumination, glossy surfaces and ambient occlusion. This implementation is tested on five different scenes and produces satisfactory approximation of indirect illumination in interactive frame rates.

Keywords: global illumination, indirect illumination, cone-tracing, voxel, octree, GPU, real-time rendering

Contents

List of Figures

List of Tables

1. Introduction

Image synthesis is rapidly developing field. Synthesis of realistic images became crucial in movie effects, animated movies, architecture visualization or video games. Simulation of physically correct light behavior is a complex task. In 1986, Kajiya published the work *The Rendering Equation* [Kaj86], where the mathematical definition of this complex task has been given. In the following decades, many methods solving this equation emerged, such as the radiosity method, described in the article *A radiosity method for non-diffuse environments* [ICG86] by Immel et al., the photon maps described by Jansen in the article *Global illumination using photon maps* [Jen96] and the path tracing desciber by Lafortune in the work *Mathematical models and Monte Carlo algorithms for physically based rendering* [Laf96].

Illumination of the scene can be decomposed into several integral parts as seen in the figure 1.1. The light bounced from the surface directly to the eye is called direct light. The light energy could be also bounced several times from the surfaces before it gets into the eye. This is called indirect light. Indirect illumination is an inseparable part of physically correct light simulation and proved to be significant for perception of the image quality, as was shown in the work by Stokes et al. [Sto+04]. The difference between the images rendered with direct illumination and with global illumination, which is composed by the direct and the indirect illumination, can be seen in the figure 1.2

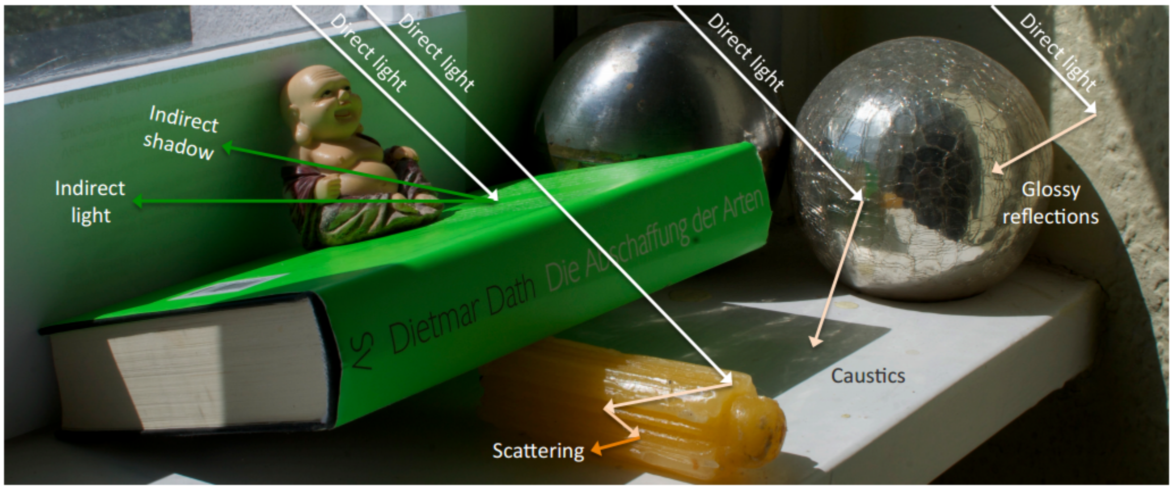


Figure 1.1: Decomposition of illumination; Courtesy of Ritschel [Rit+12]

Although the computation power of personal computers has significantly increased in the past decades, following the Moore's law, the computation of the indirect illumination, by the aforementioned methods, is still not suitable for real-time rendering of dynamic scenes.

Several methods for real-time global illumination has lately emerged. These methods usually use parallel computation on graphics processing unit (GPU). They often presents a trade-off affecting quality or/and use some sort of precomputation. This precomputation often leads to losing or affecting the dynamic property.

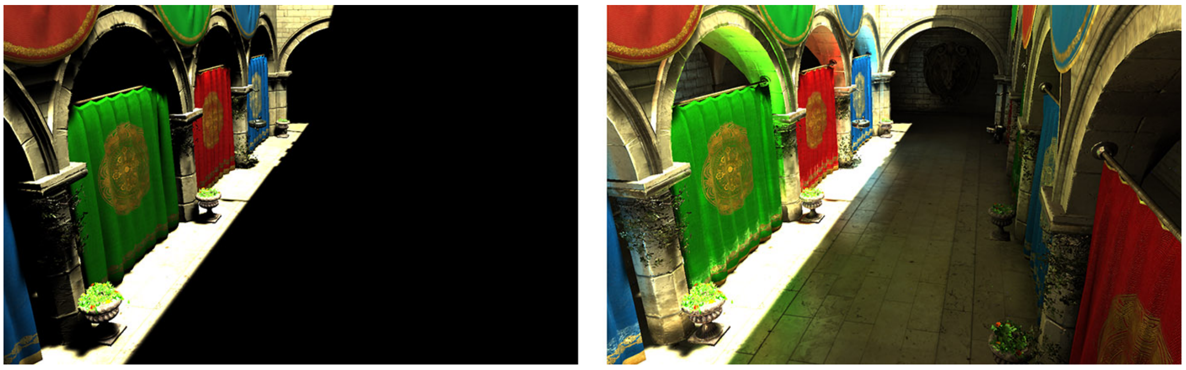


Figure 1.2: The difference between images rendered with direct illumination (on the left) and with global illumination (on the right)

1.1 Subject of this thesis

This thesis reviews several algorithms used for the real-time indirect illumination computation. The voxel cone tracing (VCT), presented by Crassin et al. in 2011 [Cra+11], is thoroughly examined and implemented. This method is designed to render second bounce of the light. It supports diffuse indirect illumination, specular indirect illumination and ambient occlusion. The light energy is stored and precomputed inside hierarchical structure. The rendering relies on the final gathering of the values from this structure. The method is capable to update this structure and thus dynamic scenes are supported. The implementation is designed to exploit capabilities of the current GPU hardware and aims for real-time rendering frame rates. The implementation is properly tested on five different scenes. The results are compared to the other algorithm designed for real-time global illumination - instant radiosity with imperfect shadow maps [Rit+11].

1.2 Thesis structure

This thesis is divided into 7 chapters. In chapter 2 we explain the theory behind the global illumination and common rendering techniques used in our implementation. Next, in chapter 3 we deliver the review of the current indirect illumination algorithms for the real-time rendering. In chapter 4 we explain why we chose voxel cone tracing and also we state the technology used for our implementation. The chapter 5 is dedicated to the detailed description of the implementation. We present several hurdles associated with the voxel cone tracing and then we present the techniques used to overcome them. In chapter 6 we present various measurements from the testing of our implementation on the several scenes and we compare the quality of the images produced by our implementation with the images produced by other algorithms. In chapter 7 we sum up the attributes of the method and our implementation and we also propose several possible improvements for a future work. The image gallery and the user manual can be found in the appendices.

2. Theoretical background

In this chapter we first explain important radiometry quantities used in global illumination calculations. We clarify the Bidirectional Reflectance Distribution Function used for shading of the surface and we examine Rendering equation. In the last part of this chapter we describe the common techniques used in computer graphics, that are related to the subject of this thesis.

2.1 Radiometry

Radiometry is the field studying and measuring electromagnetic radiation. For the illumination computations there are several important quantities: flux, irradiance, radiant intensity and radiance.

Flux Φ is defined as radient energy flowing through a surface per unit time. It's unit is Watt.

$$\Phi = \frac{dQ}{dt} \qquad [W] \tag{2.1}$$

Irradiance E is defined as incident flux per unit surface area. It's unit is Watt/m^2.

$$E = \frac{d\Phi}{dA} \qquad \left[\frac{W}{m^2}\right] \tag{2.2}$$

Radiant intensity I is defined as flux per solid angle. It's unit is Watt/sr.

$$I = \frac{d\Phi}{d\omega} \qquad \left[\frac{W}{sr}\right] \tag{2.3}$$

Radiance L is defined as flux per solid angle per unit projected area. It's unit is Watt/m^2sr.

$$L = \frac{dE}{d\omega} = \frac{d^2\Phi}{d\omega dA^{\perp}} = \frac{d^2\Phi}{d\omega dA\cos\Theta} \qquad \left[\frac{W}{m^2 sr}\right] \tag{2.4}$$

2.2 Surface representation

We perceive all the objects around us by the light reflected or refracted from these objects. We can distinguish the materials like wood,concrete or metal by the amount of the reflected light in a direction. The functions like BRDF and BTDF are used to describe this material behaviour.

2.2.1 Bidirectional Reflectance Distribution Function

Bidirectional Reflectance Distribution Function, shown in equation (2.5), is based on work of Nicodemus from 1965 [Nic65]. BRDF is used to describe the reflected light and is defined is a ratio of differential radiance reflected into the direction ω_r to the differential irradiance coming from a direction ω_i. The involved parameters are shown in figure 2.1.

$$f_r(x, \omega_i, \omega_r) = \frac{dL_r(x, \omega_r)}{dE_i(x, \omega_i)} = \frac{dL_r(x, \omega_r)}{L_i(x, \omega_i)\, cos\theta_i d\omega_i} \qquad \left[\frac{1}{sr}\right] \tag{2.5}$$

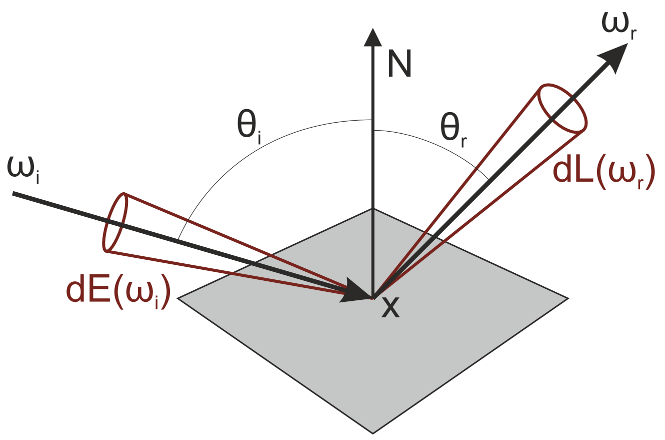


Figure 2.1: The BRDF describes how much of the incident light along the direction ω_i is scattered from the surface in the direction ω_r.

The physically based BRDF must conserve energy [PH10]. The total energy of light reflected is less than or equal to the energy of incident light, as is shown in equation (2.6). The another condition of the physically based BRDF is reciprocity. For all pairs of directions ω_i and ω_r apply $f_r(x, \omega_i, \omega_r) = f_r(x, \omega_r, \omega_i)$

$$\int_\Omega f_r(x, \omega_i, \omega_r) \, cos\theta_i d\omega_i \leq 1 \qquad (2.6)$$

2.2.2 Bidirectional Transmittance Distribution Function

The BRDF is used only to describe the amount of reflected light. The transparent materials such as glass or water also refract light. The Bidirectional transmittance distribution function (BTDF) is used to describe this phenomena. It is defined in similar manner to that of the BRDF and is generally denoted by $f_t(x, \omega_i, \omega_t)$, where the directions ω_i and ω_t are heading in opposite hemispheres around a surface point. The parameters used in BTDF are shown in figure 2.1.

2.2.3 Normal Distribution Function

Normal Distribution Function (NDF) was introduced by Alain Fournier in 1992 [Fou92]. The function expresses the density of the normals as a function of direction. Recently the NDF is used for mip-mapping normal maps [Tok05].

Gaussian function, shown in equation (2.7), is one of the possible forms of NDF.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad (2.7)$$

The gausian function's term σ^2 is called *deviation* and the term μ is called *mean*. The impact of the deviation to shape of gaussian function is shown in figure 2.3.

The product, as well as convolution, of two gaussian distributions is also a gaussian distribution [Bro03]. The mean and deviation of the product C_p, made from gaussian distributions A and B, are calculated by equations (2.8). The mean and deviation of convolution C_c, made from gaussian distributions A and B, are calculated by equations (2.9).

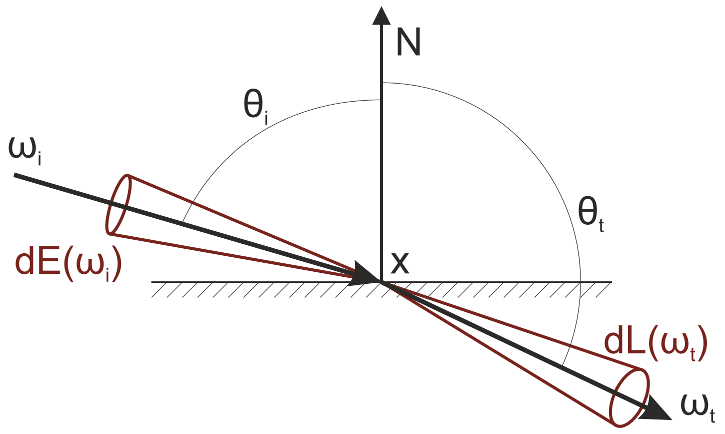


Figure 2.2: The BTDF describes how much of the incident light along the direction ω_i is refracted through the surface in the direction ω_t.

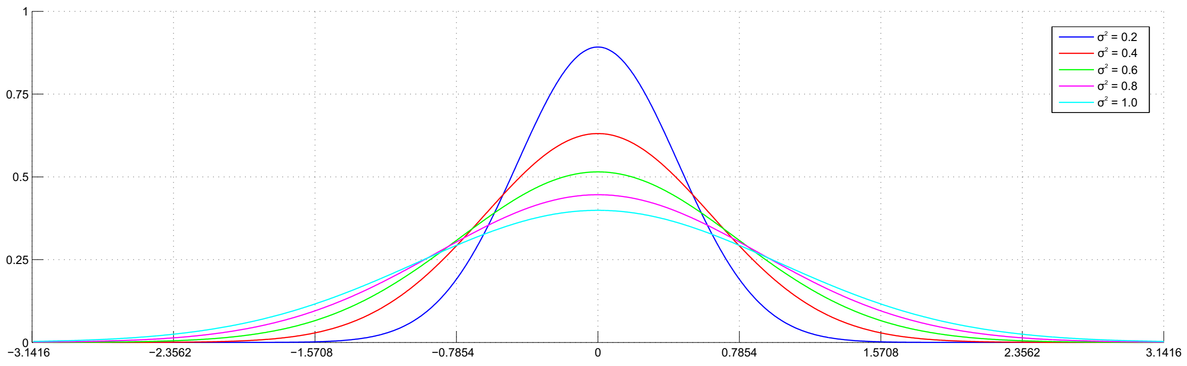


Figure 2.3: The graph of the gaussian functions with different deviations σ^2

$$\mu_{C_p} = \frac{\mu_A \sigma_B^2 + \mu_B \sigma_A^2}{\sigma_A^2 + \sigma_B^2} \qquad \sigma_{C_p}^2 = \sqrt{\frac{\sigma_A^2 \sigma_B^2}{\sigma_A^2 + \sigma_B^2}} \tag{2.8}$$

$$\mu_{C_c} = \mu_A + \mu_B \qquad \sigma_{C_c}^2 = \sqrt{\sigma_A^2 + \sigma_B^2} \tag{2.9}$$

The gaussian function can be used as directional representation. In that case, the mean is an average vector D of the gaussian lobe. As described in the *Mipmapping normal map* by Toksvig [Tok05], the value σ can be computed from the length of average vector D using the equation (2.10). That allows to create gaussian lobes from 2 or more vector to represent their common direction. The examples of the gaussian lobes created from various vectors in 2D are shown in figure 2.4. The gaussion lobe is determined by the vectors used for its computation. On the other hand, the gaussian lobe does not determine these vectors. In 3D space, the identical gaussian lobe can be made from any number of vectors as is shown in figure 2.5.

$$\sigma = \sqrt{\frac{1 - |D|}{|D|}} \tag{2.10}$$

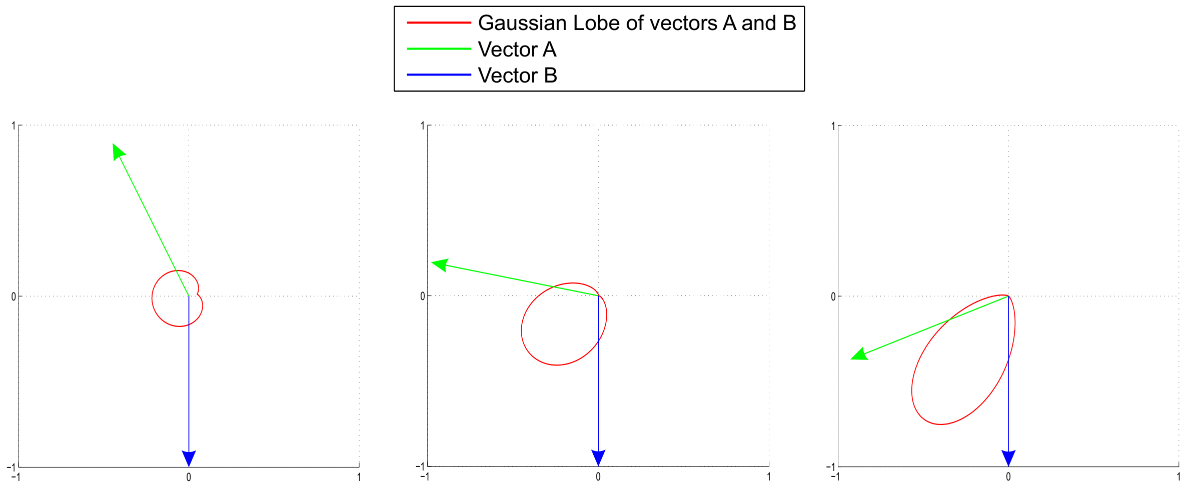


Figure 2.4: The shape of the gaussian lobe changes with angle between the vectors.

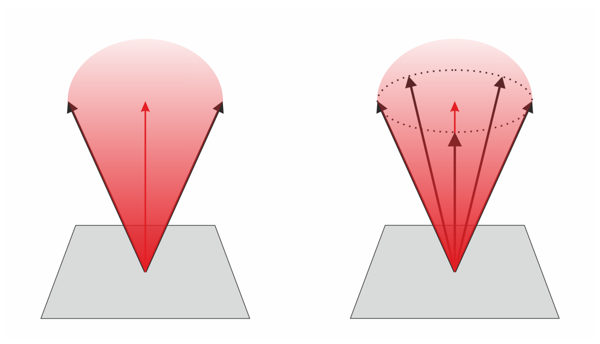


Figure 2.5: Gaussian lobes on the left and on the right have same shape, although they were derived from different vector sets.

2.3 Rendering equation

The recursive rendering equation (2.11), introduced by Kajiya [Kaj86], express the outgoing radiance of the point x in direction $\vec{\omega}_r$.

$$L\left(x, \vec{\omega}_r\right) = L_e\left(x, \vec{\omega}_r\right) + \int_{\Omega} f_r\left(x, \vec{\omega}_i, \vec{\omega}_r\right) L\left(h\left(x, \vec{\omega}_i\right), -\vec{\omega}_i\right) cos\Theta_i d\omega_i \tag{2.11}$$

The radiance is a sum of the light emitted by the surface itself L_e and the reflected radiance. The function $h\left(x, \vec{\omega}\right)$ returns the position of the first hit when shooting a ray from position x into direction

$\vec{\omega}$. The function f_r expresses the surface properties. This function is the BRDF, explained in section 2.2.1. The θ_i stands for the angle between the $\vec{\omega}_i$ and the normal of the surface. As the angle increases, the projected area, expressed as $cos\theta_i$, decreases.

The result of the BRDF function f_r in the rendering equation (2.11) depends on incident direction $\vec{\omega}_i$ and the reflection direction $\vec{\omega}_r$. In the radiosity setting the f_r function is restricted only to diffuse reflection $f_r = \frac{\rho_d(x)}{\pi}$. The rendering equation then becomes isotropic (2.12). The term $\rho_d(x)$ is called albedo. Albedo is defined as the ratio of reflected radiance from the surface to incident radiance upon it.

$$L(x) = L_e(x) + \frac{\rho_d(x)}{\pi} \int_{\Omega} L\left(h\left(x, \vec{\omega}_i\right)\right) cos\Theta_i d\omega_i \tag{2.12}$$

2.4 Common rendering techniques

In this section we explain common techniques used in image synthesis that are used in our implementation. First we explain normal mapping, which allows to keep shading details of the high resolution mesh on a low resolution mesh. Next we will describe the shadow mapping and then we describe the octree, which is the essential structure for the implemented algorithm. Another important technique for our implementation is deferred rendering.

2.4.1 Normal mapping

As was shown in section 2.2.1, the illumination of a single surface point is dependent on the surface normal. The triangle can have either one normal for the whole it's surface, or three normals, one for each vertex. In the second case the normal of a surface point on the triangle is tri-linearly interpolated from these 3 vertex normals.

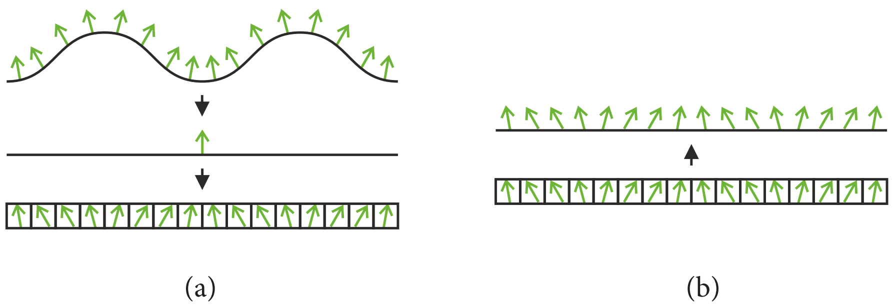


Figure 2.6: The normals of the high resolution mesh are projected onto the low resolution mesh and are stored in the normal map (a). During the rendering, the normal of a point on low resolution mesh is altered by the normal acquired from normal map (b).

In real-time computer graphics there is the need to keep triangle count of 3D objects as low as possible, to reduce the cost of vertex transformations. The normal mapping, based on the work of Krishnamurthy et al. from 1996 [KL96] and introduced by Cohen et al. in 1998 [COM98], is used to mimic the shading behaviour of a high resolution object with low resolution object, which has significantly lower count of triangles, and a texture. This texture is called *normal map*. Each pixel of the normal map corresponds with a surface point of the low resolution object. The high resolution object's surface is projected onto the low resolution object and for each pixel of the normal map, the normal of high resolution mesh is stored. During the rendering, the normal of a triangle is altered or completely replaced by a normal acquired from the normal map. The illustration of normal mapping is shown in figure 2.6.

2.4.2 Shadow mapping

To solve the rendering equation, which is explained in section 2.3 it is necessary to compute many visibility tests between a surface point and a light. The Shadow Maps, presented in 1978 in *Casting curved shadows on curved surfaces* by Williams [Wil78], presents a simple solution of such visibility tests.

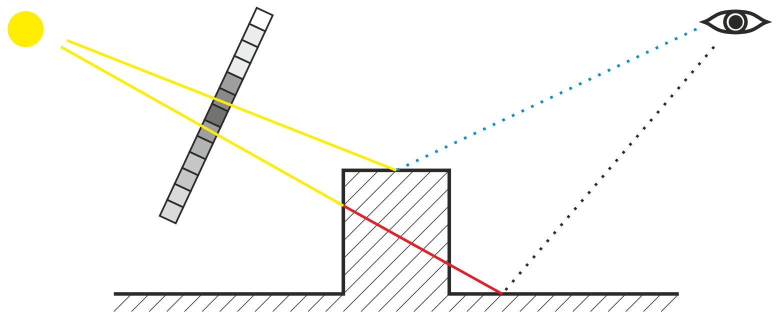


Figure 2.7: The depth stored in the shadow map (yellow) is compared with the distance of the surface from the light (red). If the distance from the light is greater than the depth from the shadow map, then the surface lies in a shadow.

The scene is projected to a texture from the position and the direction of a light. The projection is computed using a projection matrix P_l. For each pixel of a texture the depth of the front most surface (in the meaning of projection direction) is stored.

Once the image is being rendered from the view of a camera, for each rendered surface point with world space position p_{ws}, the position in the shadow map p_{sh} is computed using equation (2.13). If the distance of the surface point from the light is greater than the value stored in the shadow map, than the surface point lies in the shadow. The illustration of shadow mapping is shown in figure 2.7.

$$p_{sh} = P_l * p_{ws} \tag{2.13}$$

The shadow mapping is simple and effective solution but it has two big issues. The resolution of shadow maps is limited. When this resolution is small or the shadow map is projected on large area, the shape of the pixels is visible. Percentage Closer Filtering (PCF), introduced by Reeves et al. in 1987[RSC87], addresses this issue and significantly improves the quality of the shadow mapping. When the PCF is used, the occlusion of a surface point is computed from N samples acquired from a shadow map. The total occlusion is the average of these sample values.

The limited resolution of a shadow map means that it stores a discretization of the scene depth. Moreover the depth value is stored in limited precision. Both these facts cause shadowing artefacts shown in figure 2.8. This issue can be solved with careful choose of the bias that is used when the distance from shadow map is compared with surface distance.

2.4.3 Octree

In real-time computer graphics, the most common representation of the 3D object is a set of triangles. The 3D models represented in other means, for example as a point set or by curved surfaces, are often converted to set of triangles before they are rasterized.

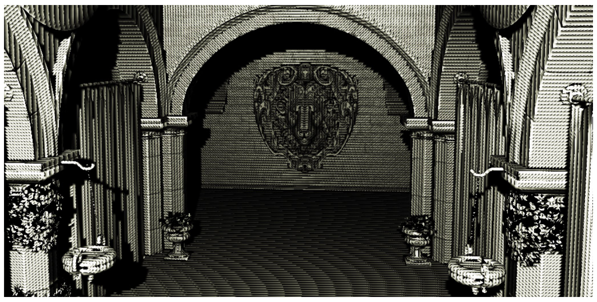


Figure 2.8: The shadow mapping with no bias added to the shadow map depth is causing sampling artefacts

In order to decrease the computational complexity of the processes like a ray-tracing or a frustum culling, the scene elements are often arranged in a hierarchical structure. One of the hierarchical structures is the octree. The octree structure, which was introduced by D. Meagher in 1980 [Mea80], is a spatially dividing tree structure. Example of a octree build around the mesh is given in figure 2.9.

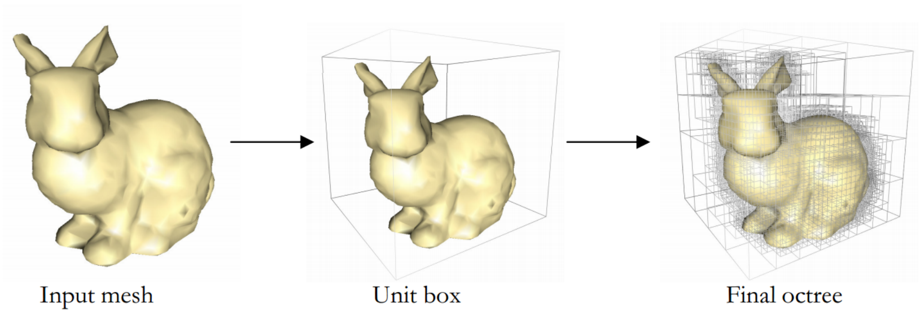


Figure 2.9: Building of an octree around the mesh surface; Courtesy of Lefebvre et al. [LHN05]

Each internal node has exactly 8 children or leaves, and is dividing the space, it represents, into 8 octants. The space with no elements is not further divided. The illustration of the octree is shown in figure 2.10.

If an octree is used to represent a scene composed by triangles, then these triangle are stored in the leaves of the octree. When we use this octree to perform ray-triangle intersection test, then the nodes hit by the ray are visited in depth-first search (DFS) manner. The triangles located in the nodes not hit by a ray are discarded from computation.

In current real-time applications a object's surface is often described using several texture maps. Even the geometry of the object is altered using displacement maps. As the memory requirements of this representation increase, the other representations are becoming more profitable.

The octree structure can be also used as a representation of the scene. Such octree is called sparse voxel octree [LK10]. Each node of such octree is a voxel with its own parameters such are color, normal or opacity. Time complexity of rendering a single pixel of an image with such representation is O(log N), where the N denotes the number of voxels.

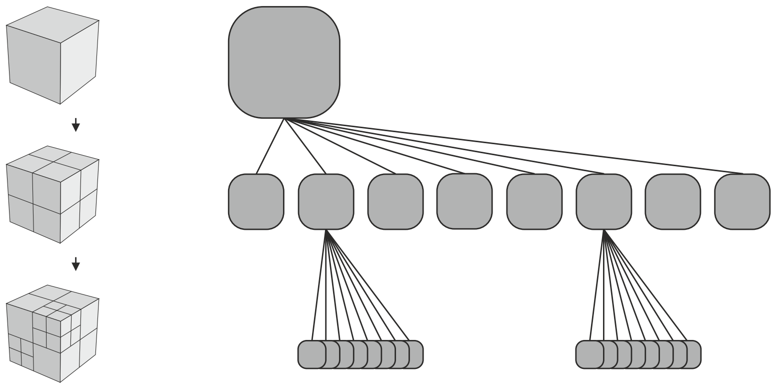
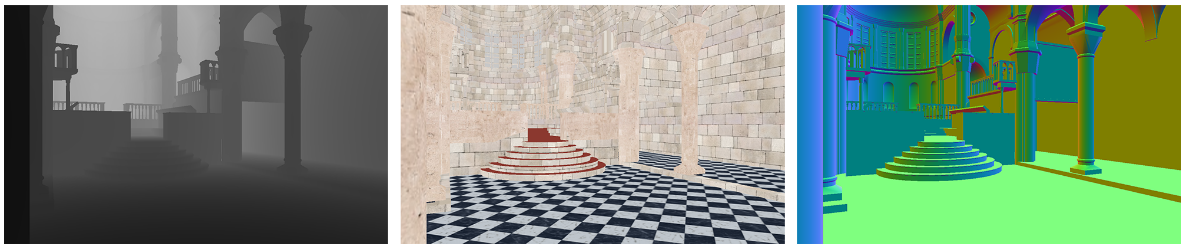


Figure 2.10: Each inner node of the octree divide the space it represents into 8 sections.

2.4.4 Deferred rendering

In traditional forward rendering the scene geometry is rasterized and per-pixel operations, such is shading and lighting, are performed on each fragment generated during this rasterization. This concept is infective when the part of the scene is frequently overdrawn by the objects closer to a camera.

It is often appropriate to compute majority of the computations only on the visible fragments of the scene. It is possible by the concept called *deferred rendering*, based on the work by Deering et al. from 1988 [Dee+88]. The scene is rendered only once and various attributes of the scene are stored in a buffer. This buffer is called Geometry buffer (g-buffer) and was introduced by Saito et al. in 1990 [ST90]. The typical content of the g-buffer is the depth of the scene, albedo, and normal. The example of such g-buffer content is given in figure 2.11. Then, in another pass, the values from g-buffer are used to compute shading, lighting and possibly another computation on visible fragments only.



(a) (b) (c)

Figure 2.11: The content of the geometry buffer: depth of the scene (a), albedo (b) and decoded normals (c)

The deferred rendering has several drawbacks. The geometry buffer consumes great amount of GPU memory and memory bandwidth. To minimize this issue it is necessary to pack and compress g-buffer attributes, so they use less memory. This issue is becoming less relevant with the increasing size of current GPU's memory.

As all Z-buffer rendering algorithms the deferred rendering is not capable of correct rendering of transparent objects. The order independent transparency is often solved using depth peeling technique, introduced by Everitt in 2001 [Eve01].

3. Interactive Global Illumination Algorithms

In this chapter we review several algorithms designed for real-time rendering. The last reviewed algorithm, Indirect Illumination using Voxel Cone tracing, is described in detail. The implementation details of this algorithm are to be found in chapter 5.

3.1 Virtual Point Lights

Many global illumination algorithms use concept of Virtual Point Light (VPL). This concept was first introduced by Keller in *Instant Radiosity* [Kel97].

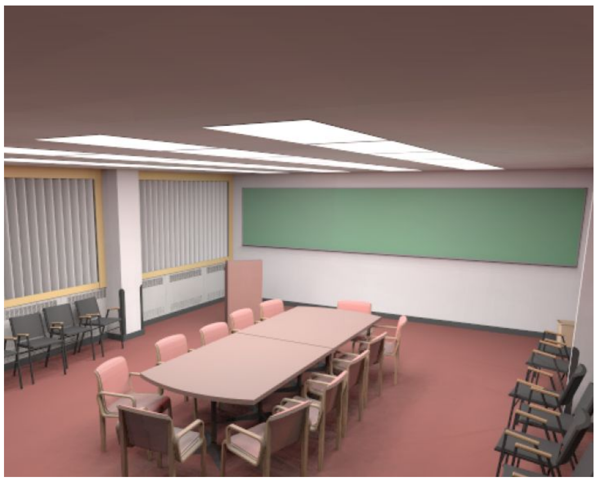


Figure 3.1: The rendered image of the conference room using 128 virtual point lights; Courtesy of Keller [Kel97]

Using this radiosity setting shown in equation (2.12) and the quasi-Monte Carlo integration, they approximate radiance in the scene by a set of virtual lights. Radiance coming to a point y is approximated by the radiance coming from the virtual point lights. The visibility tests for each such light are performed using common shadowing techniques and the radiance coming from them is accumulated in a accumulation buffer.

Virtual point lights are generated from particles shot by the primary lights using Halton sequence sampling. At first, N particles are generated. Since not 100% of the radiance is absorbed, some particles are reflected. After the first hit, $\overline{\rho}N$ particles are reflected. After $j-1$ reflections $\overline{\rho}^jN$ particles are reflected. The $\overline{\rho}$ is defined according to equation 3.1, where the scene is composed of K surface elements A_k with average reflectivity $\rho_{d,k}$.

$$\overline{\rho} := \frac{\sum_{k=1}^{K} \rho_{d,k}|A_k|}{\sum_{k=1}^{K} |A_k|} \tag{3.1}$$

The number of all generated particles M is linear in N, as shown in equation 3.2.

$$M < \sum_{j=0}^{\infty} \overline{\rho}^j N = \frac{1}{1-\overline{\rho}} N =: \overline{l} N \tag{3.2}$$

The image rendered with above described method is shown in figure 3.1.

3.2 Reflective Shadow Map

Another concept used in some global illumination algorithms is Reflective Shadow Map. Reflective Shadow Maps (RSM) were introduced in 2005 by Dachsbacher et al. [DS05]. The idea is based on using shadow maps as the source of the first bounce of the indirect light. Shadow maps are used to determine the visibility of a light source from a surface point. To obtain a shadow map, they render visible surfaces of the scene from the light source. These visible surfaces are in fact the only surfaces involved in the second light bounce. In Reflective Shadow Map each pixel is considered to be a source of the light. For each pixel p we need to store depth d_p, position x_p, normal n_p and the reflected radiant flux Φ_p. To archive this RSM uses multiple render targets. The content of these render targets is shown in figure 3.2.

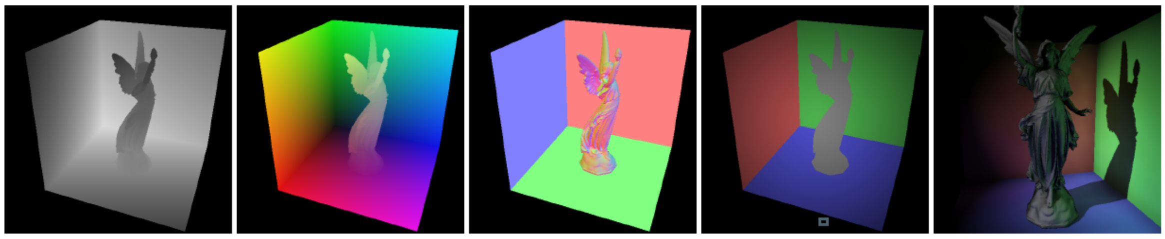


Figure 3.2: The content of the RSM. From left to right: depth, world space position, normal, flux. On the right there is the image rendered with global illumination using RSM; Courtesy of Dachsbacher et al. [DS05]

If we assume that all surfaces are diffuse reflectors, the radiant intensity emitted into direction ω from a pixel of RSM is described by equation (3.3). Where Φ_p is the flux of the pixel and n_p is the normal of the pixel.

$$I_p(\omega) = \Phi_p max\left\{0, n_p \cdot \omega\right\} \tag{3.3}$$

The original article from 2005 [DS05] also describes how to use such RSM to illuminate a surface. The indirect illumination of surface point is computed from the sum of intensities of all visible RSM pixel lights. It is computationally expensive to compute this even for RSM of resolution 512×512 pixels. Instead the indirect illumination is computed only from several samples of RSM. The position of illuminated surface x_p is projected into RSM. The samples are selected around this projected position. The density of the samples is decreasing with square distance to the projected position. This assumes that two near projected surfaces in RSM are also near in RSM. Another assumption is that the sample is directly visible from illuminated surface. This two assumptions can lead to wrong results in indirect illumination.

3.3 Cascaded Light Propagation Volumes

Cascaded Light Propagation Volumes, presented by Kaplanyan et al. in 2010 [KD10], is another real-time indirect illumination algorithm. The examples of image rendered using Cascaded Light Propagation Volumes are given in figure 3.3.

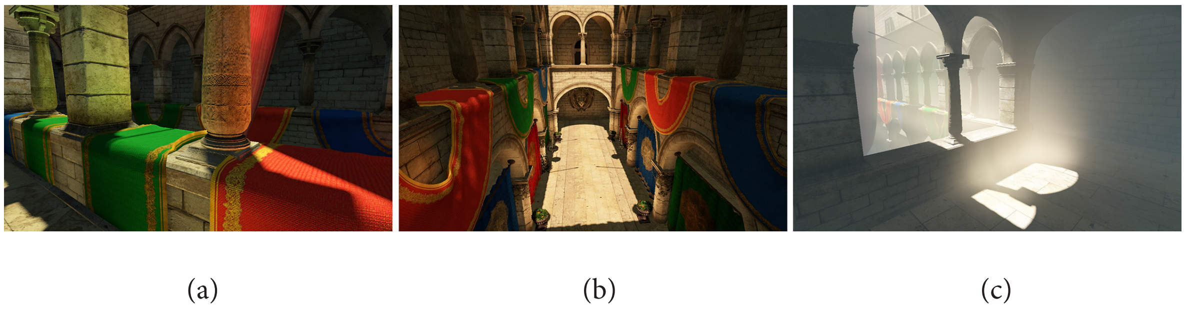


Figure 3.3: The Sponza scene rendered using the Cascaded Light Propagation Volumes; the LPV allows to render participating media such are fog, water or smoke (c); Courtesy of Kaplanyan et al.[KD10]

The method simulates the light transport using techniques similar to grid-based fluid simulation algorithms [CLT07]. The light intensity is stored inside a grid and iteratively, each cell transfers light intensity to its neighbours. This grid is called *Light Propagation Volumes* (LPV). The light is blocked by the geometry of the scene, which is sampled and stored in another grid called *Geometry Volumes* (GV). To archive better performance and lower memory consumption, the set of nested grids is used. For objects closer to the eye the indirect illumination is computed using finer grid. Illustration of the nested grids is shown in figure 3.4.

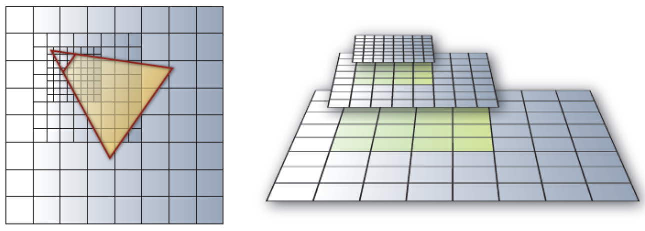


Figure 3.4: The grids are overlapping. The finer grids are translated according to camera position and direction. Courtesy of Kaplanyan et al. [KD10]

First, for every light source, they render a reflective shadow map. Each texel of the RSM is considered a VPL. The VPL's intensity is accumulated and stored as spherical harmonics (SH) inside the cells of the grid.

For correct light propagation, the algorithm needs convenient access to scene geometry. Similarly to the light intensity, they store volumetric representation of the scene in the grids. This representation is stored in Geometry Volumes (GV). GV are displaced by half the cell size with respect to the LPV.

This ensures that the GV cell centers are in the corners of the LPV cells. For acquiring scene samples, they utilizes the geometry buffer from a camera view and the light source's RSMs. It is also possible to reuse valid samples from previous frames. Samples' normals are stored the same way as light intensity, as spherical harmonics.

Light propagation is implemented as an iterative process. The light intensity is in each iteration propagated to its 6 neighbours along main axial directions. First, for each adjacent cell, the flux incident to each of the cell's faces is computed. Figure 3.5 illustrates a 2D example of this process. As a next step, the incident flux of each cell is transformed into outgoing intensity. They archived this by creating VPLs, each facing one face of the cell and emitting flux equal to the flux of the face. Subsequently, these VPLs are accumulated into the intensity grid and stored as new spherical harmonics, using same process as in initial light injection step. Whenever the light is propagated from the source cell to the destination cell, the light is attenuated by a occlusion computed from bi-linearly interpolated SH-coefficients of the GV.

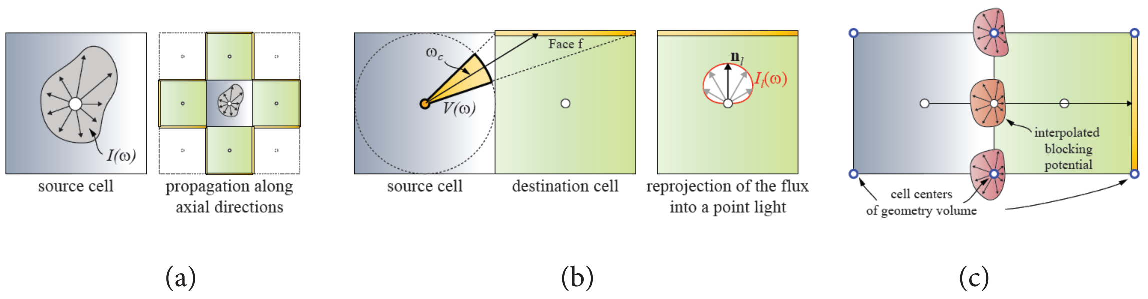

I(ω)
source cell
propagation along axial directions
(a)
ω_c
V(ω)
Face f
source cell
destination cell
n_l
I_l(ω)
reprojection of the flux into a point light
(b)
interpolated blocking potential
cell centers of geometry volume
(c)

Figure 3.5: Propagation of the light intensity is performed between axial neighbours (a); Incident flux is computed for each of the destination cell's face (b); GV cell centers are located in the corners of LPV cells (c); Courtesy of Kaplanyan [KD10]

After ι iteration of the light propagation step, the LPV represents light distribution in the scene. For illuminating a surface point, they obtain the light intensity by tri-linear interpolation of the SH coefficients from nearest grid cells. The intensity must be then converted into incident radiance.

3.4 Hardware-Accelerated Image Space Photon Mapping

Image Space Photon Mapping was developed by McGuire et al. in 2009 [ML09]. They accelerates certain steps of traditional photon mapping [Jen96] by GPU. The example of image rendered using ISPM is given in figure 3.6.

3.4.1 Initial Bounce

The initial tracing of photons from the light source to the first surface hit is expensive, since the most photons must be traced during this step. The rays used to trace these photons are coherent, they have common origin. This initial tracing can be implemented as a projection and this projection can be efficiently computed using GPU rasterization. Omni-directional lights are rendered into *bounce cubemap*. A fragment shader scatters the photon travelling through each pixel of the bounce cubemap by surface properties. The properties of the surface are world-space position, normal and possibly parameters of bidirectional scattering distribution function (BSDF). BSDF is the combination of the BRDF and BTDF and is used to describe the reflected and refracted light.

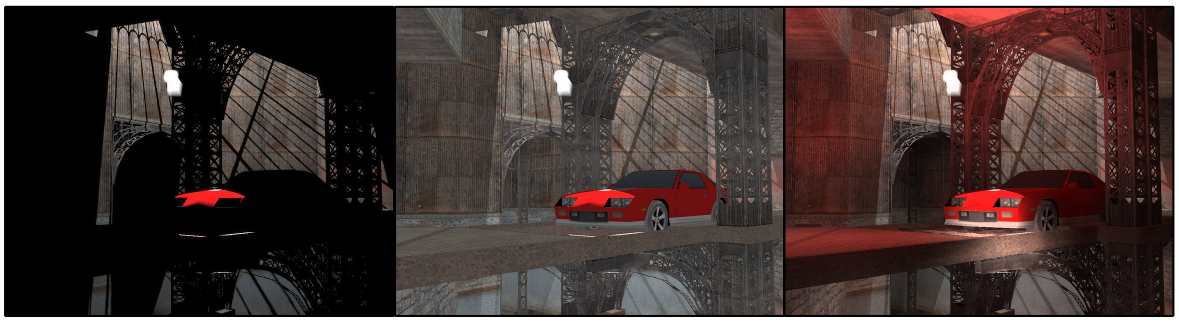


Figure 3.6: Image rendered with direct illumination (a), image rendered with direct illumination and constant ambient light (b), image rendered using ISPM (c) (the intensity of ambient and indirect illumination are amplified); Courtesy of McGuire et al. [ML09]

3.4.2 Secondary Bounces

Secondary bounces of the photons, unlike the initial bounce, does not have common projection center. Each photon from bounce cubemap is bounced into the scene in the direction defined by surface BSDF. Using rasterization pipeline of the GPU is in this case unsuitable. They rely on CPU-based approach instead.

Each photon is traced using ray-tracing. Since ray-tracing involves many ray-triangle intersection tests, it's necessary to use a accelerating structure. They propose a hybrid structure combining kd-tree and bounding sphere hierarchy (BSH), which is a special case of bounding volume hierarchy (BVH). The kd-tree for static part of the scene is computed once in preprocess. The dynamic part of the scene is approximated with the BSH composed entirely from spheres each frame. Such hybrid structure is well suited for partially dynamic scenes. The coarse dynamic part approximation affects only the secondary and higher bounces, which are often diffuse. Thus the impact on the indirect illumination quality of such approximation is often imperceptible.

3.4.3 Radiance estimate

To estimate the radiance of the surface point it is necessary to obtain nearby photons. Their energy is then weighted by a filter kernel κ. Traditionally it is expensive, since for each surface point the neighbours in the diameter d have to be found. In ISPM they use different approach. The radiance is scattered from the photons to the nearby surface. The scattered radiance is weighted by the same filter kernel κ.

For each photon they generate a photon volume. This volume bounds non-zero portion of the kernel κ. All these photon volumes are rendered using GPU rasterization and contribution of photons radiance is additively blended with direct illumination image. The photon volumes and its contribution to surface illumination are shown in figure 3.7.

3.5 Imperfect Shadow Maps

In 2008 Ritschel introduced *Imperfect Shadow Maps for Efficient Computation of Indirect Illumination* [Rit+08]. The method is used to speed up Instant Radiosity visibility computations. Commonly, the

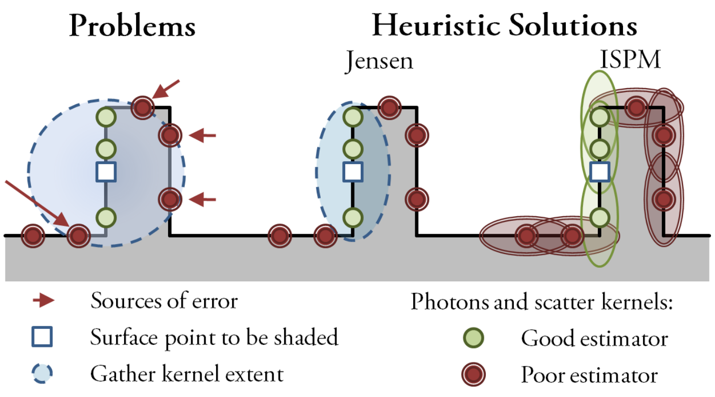


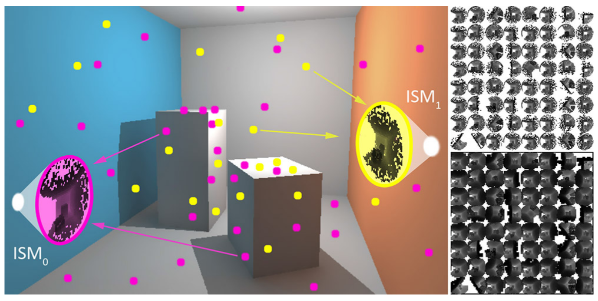
Figure 3.7: Spherical shape of the kernel can lead to illumination artifacts. The heuristic solution by Jensen [Jen96] compresses the kernel along the the normal to the shading surface, ISPM compresses the kernel along the normal of a surface hit by photon. Courtesy of McGuire [ML09]

mutual visibility of a point and a light source is computed accurately by ray-casting or traditional shadow map. Imperfect shadow maps (ISM) are small shadow maps, e.g. 32×32 pixel, that solve otherwise expensive visibility tests for Virtual Point Lights.

First, the RSM is generated from all sources of light. Each texel of the RSM is potential VPL (pVPL). A pVPL is selected to be VPL using a importance sampling based on outcoming radiance. For each of these VPLs the ISM is rendered.

Rendering of the whole scene is needless and expensive for ISM. They approximate the scene by a point cloud. Each point is created by a random selection of a triangle, with a probability proportional to the triangle area, and then by selecting a random position on that triangle.

These points are scattered to random ISM's depth buffer during ISM rendering. Non-continuous point representation, as well as the scattering of the points, lead to holes in the ISMs. These holes are filled using a push-pull post-processing introduced by R. Marroquim et al. [MKC07]. These imperfections as well as the low resolution of ISMs lead to inaccurate visibility information. This inaccuracy is not noticeable when a large number of the VPLs is used. The ISMs are packed into texture atlas. This allows the ISM rendering to be computed using single draw call on GPU. Point scattering and the texture atlas of ISMs is shown in figure 3.8.



(a) (b)

Figure 3.8: Geometry point samples are scattered randomly into two ISMs (a). Texture atlas composed of ISMs.(b); Courtesy of Ritschel [Rit+08]

Ritschel et al. indroduced View-Adaptive Imperfect Shadow Maps in 2011[Rit+11]. Images rendered using View-Adaptive Imperfect Shadow Maps are shown in figure 3.9.

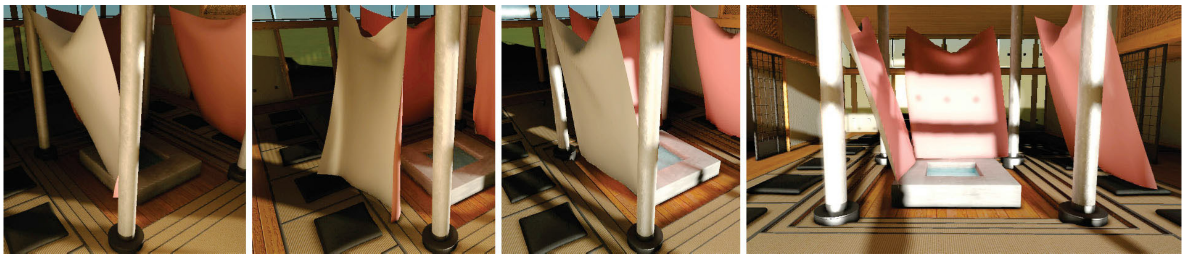


Figure 3.9: Images rendered using View-Adaptive Imperfect Shadow Maps; Courtesy of Ritschel [Rit+11]

View-Adaptive Imperfect Shadow Maps are improvement of the original Imperfect Shadow Map algorithm, which has two shortcomings. To produce plausible indirect illumination, it is necessary to create many VPLs. These VPLs are selected without considering the contribution on the final image. To overcome this, they use Bi-directional Reflective Shadow Map (BRSM). For each texel of this RSM, which is in fact potential VPL, they evaluate the contribution on the view. This impact is evaluated from a few randomly-chosen view-samples. The visibility test during this evaluation is neglected. Each pVPL's average contribution is stored in BRSM. The VPL set is then chosen using importance sampling based on this bi-directional importance.

Another shortcoming of the original ISM is that a regular point-based geometry representation could be too coarse in large scenes and it can lead to worse ISM artefacts. In Adaptive ISM the geometry point samples density is dependent on the projection of the geometry to the view samples. For each triangle this projection is tested with few random view samples. The geometry further from the view has often small impact on the final image and therefore is sampled roughly. The geometry closer to the view has often large impact on the final image is sampled densely.

3.6 Indirect Illumination using Voxel Cone Tracing

Indirect Illumination using Voxel Cone Tracing was introduced by Crassin et al. in 2011 [Cra+11]. They use a sparse octree structure of voxels to pre-filter values used for indirect illumination. The sparse octree structure requires less memory because only the actually used voxels are stored.

The algorithm consists of 3 steps. First, the light and scene parameters are stored inside the leaves of the voxel octree. Next, these parameters are filtered inside the octree. In the last step, for each visible surface point, these values are gathered over a hemisphere. This gathering is often made by ray-tracing of many rays. The traced rays over the hemisphere are directionally and spatially coherent. The Voxel Cone Tracing (VCT), which they have used for gathering the light parameters, is designed to exploit this coherency.

3.6.1 Voxel-Octree building

The algoritm is designed for use with dynamic scenes. Although the scene is divided into static and dynamic part, the voxelization process must be fast, because the dynamic part of the scene is updated in each frame. The method they proposed for the voxelization exploits hardware rasterization pipeline and is fast enough to handle voxelization of the dynamic part of the scene in real-time frame rates.

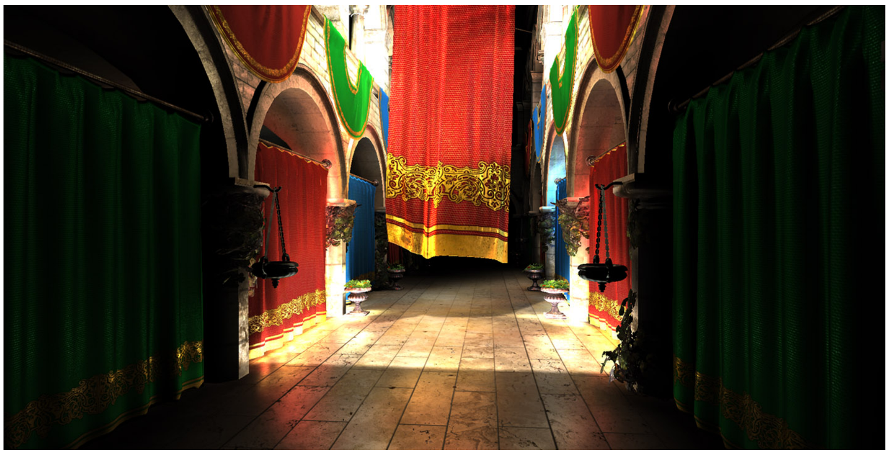


Figure 3.10: Image rendered using Voxel Cone Tracing; courtesy of Crassin et al. [Cra+11]

At first, the scene is rendered using orthographic projection and without using Z-buffer. Each triangle is projected using one of three main axis according its normal. Each fragment produced with such projection is inserted into a *voxel-fragment list* with its parameters like a world-space position, normal and texture color.

In the next step, they generate a thread for each of the fragments in the *voxel-fragment list*. The fragments are traversed down the octree, which is initially only a root node. Every time the leaf needs to be subdivided, new node is created and stored in pre-allocated video memory. The position of the created node in the video-memory is determined by an atomic counter, which is incremented with each new node. Many threads can be involved in subdividing a node, especially in shallow depths of the octree. To avoid conflicts between the threads, each node is associated with a mutex. The subdivision is performed only by the first thread, while other threads are interrupted. To avoid an active waiting loop in GPU threads, which would be expensive, the interrupted threads are placed in *global thread list*. At the end of the each division step, the threads inside *global thread list* are re-run. The octree construction is complete once the *global thread list* is empty.

3.6.2 Voxel representation

The algorithm is designed to exploit hardware tri-linear filtering. Since two adjacent nodes of a sparse octree are generally not located consequently in the memory, the hardware tri-linear filtering would not be possible to use with a single value per parameter of the node. Instead the node contains a *brick*. Brick is the lattice of 3^3 cell, where the cells are located in the corners of the node's children as shown in the figure 3.11. The border cells of the neighbours are identical.

The voxel is represented by several parameters. These parameters are diffuse color, opacity, normal, light direction and light intensity. Light direction and normal are stored as isotropic Gaussian lobe.

3.6.3 MIP-mapping

Each of this parameters is initially stored inside the leaves and then iteratively MIP-mapped from the lower levels to the higher levels of the octree. Each cell of the brick is filtered from 3^3 cell from the lower bricks. In order to prevent duplicitous computation, they perform this filtering in 2 steps.

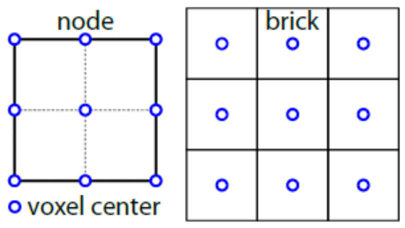


Figure 3.11: Brick values are located in the corners of the node's children. Courtesy of Crassin et al. [Cra+11]

First, for each cell they sum up the values, coming from the sub nodes of the current level node. The summed values must be weighted with the inverse of its multiplicity. This leads to 3^3 Gaussian weighting kernel, which is shown in figure 3.12.

Next, in the second step, the border cell values are transferred and averaged to neighbouring bricks. They perform this transfer in six passes. First pass will add border values to corresponding values of the negative x neighbour. Second pass will copy border cell values, now containing the sum, to corresponding values of the positive x neighbour. Same process is repeated for y and z-axis. Value transfer is illustrated in the figure 3.12.

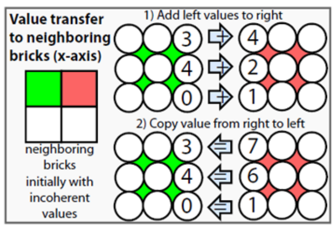

Value transfer to neighboring bricks (x-axis)
neighboring bricks initially with incoherent values
1) Add left values to right
3
4
0
4
2
1
2) Copy value from right to left
3
4
0
7
6
1

Figure 3.12: The process of value transfer to neighbouring brick along the x-axis; Courtesy of Crassin et al. [Cra+11]

3.6.4 Voxel Cone Tracing

Once the sparse voxel octree is constructed and filled with the scene and light parameters, it is used to compute the indirect illumination. Using a deferred renderer they obtain a set of visible fragments. For each such fragment, a set of cones is generated. Cone directions and apertures are determined by the BRDF of the rendered material. The Phong BRDF can be decomposed into wide diffuse lobes and a specular lobe. The visualization of Phong BRDF is shown in figure 3.13. For the diffuse lobe, several wide cones are generated over a hemisphere and for the specular lobe, the tight cone is generated. Specular cone is directed in reflected direction R.

The algorithm uses approximate voxel cone tracing for the gathering of incoming light intensity. Instead of true tracing, the values are gathered from several samples along the cone direction. For each sample a lookup into the voxel octree is made. The level of the octree used for the lookup is determined by the radius of the cone in sample location. Illustration of this process is shown in figure 3.14.

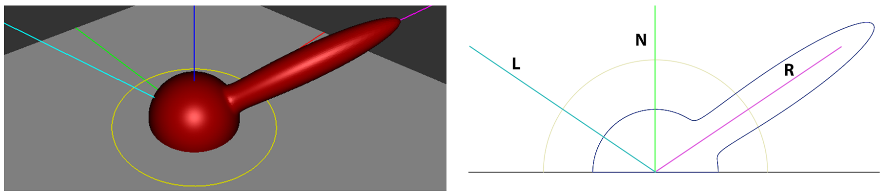


Figure 3.13: Visualisation of Phong BRDF; Specular highlight is located along the reflected direction R.

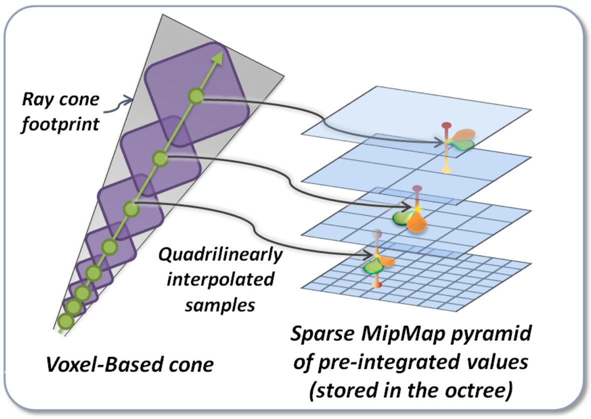

Ray cone footprint
Quadrilinearly interpolated samples
Voxel-Based cone
Sparse MipMap pyramid of pre-integrated values (stored in the octree)

Figure 3.14: The sample radius corresponds to the depth of the octree. Courtesy of Crassin et al. [Cra+11]

4. Problem Analysis

In this chapter, we will first discuss some of the algorithms explained in chapter 3 and we will explain the reasons, why we have chosen the Voxel Cone Tracing. Then, in the section 4.2, we will discuss the technology choices we have made.

4.1 Algorithm selection

The indirect illumination is a computationally expensive task. Many researchers have come with different solutions to compute it. Each of this solutions has a set of positive and negative features. This set of features often predetermine the usage of such a solution. For our solution we need a fast method, capable of rendering dynamic scenes.

Image Space Photong Mapping [ML09], unlike the other mentioned indirect illumination algorithms, is able to render not only the reflected light, but also the refracted light. This is very beneficial for the scenes containing water, where the refracted light can form caustics. The disadvantage of this method is a need to traverse the multiple bounces of the photons. This is computationally expensive step, which needs to be accelerated by a hierarchical structure. This structure must be both efficient and easy to rebuild/update, since one of our requirements is the support for dynamic scenes.

Imperfect Shadow Maps [Rit+08], as one of the algorithms used with Instant Radiosity, does not need any additional hierarchical structure for the scene. Instant Radiosity quality increases and the performance decreases with the increasing number of the generated VPLs. Visibility tests for these lights are approximated by imperfect shadow maps. This may lead to leaking of the light through walls. Instant Radiosity produces plausible static images even for lower count of the VPLs. The disadvantage of the ISM is its poor temporal stability. When the primary light moves, the VPLs are replaced and a flickering of the indirect illumination appears. Another disadvantage of all the algorithms of Instant Radiosity family is that they are able to render only the diffuse indirect illumination.

The Light Propagadition Volumes [KD10] is an unorthodox method of computing indirect illumination by light transfer, computed localy between the cells of the grid. The method is very fast and was proven to be suited for realtime rendering. The main factor affecting the performance of this method is number of local light transfer iterations. This method was implemented in CryEngine 3 [Kap09] made by Crytek. This method produces the diffuse indirect illumination only. The indirect illumination produced by this method is very coarse approximation and may produce leaking of the light.

For our solution of indirect illumination, we have chosen the Voxel Cone Tracing. The Voxel Cone Tracing, in addition to the diffuse indirect illumination, facilitates the rendering of glossy surfaces. The big advantage of this method is that the light intensity values are pre-filtered and stored in the octree. This makes the rendering of a single pixel of the final image fast. It is very scalable method. The performance can be tuned by scaling the level of the voxel octree or by adjusting the properties of the cone tracing. All the steps of the algorithm are suitable for GPU computing. It also supports dynamic scenes and therefore the Voxel Cone Tracing fulfils all our requirements.

4.2 Technology selection

The current GPU hardware offers many unified cores, which are suited for parallel tasks. Although the GPU cores have a limited functionality (in comparison with the current CPU's), the combined computational power they offer is immense. Many processes of the selected algorithm are suited for parallel computation and it is beneficial to use the power of GPU computing.

For real-time rasterized computer graphics, there are 2 widely used application interfaces - OpenGL and Microsoft DirectX. For our implementation it is essential to read from and write to random place in the GPU memory.

OpenGL [OGL4], maintained by the Khronos Group, is the cross-platform graphics library. It offers functionality strictly related to rasterized 3d computer graphics. OpenGL in version 4.2 allows direct access to memory by the *imageStore*, *imageLoad* and others image related functions.

Microsoft DirectX [DX13] is a collection of APIs to provide functionality useful for creating multimedia programs. DirectX contains APIs for handling 3D graphics, mathematical operations, sound, input and others. DirectCompute, present in DirectX 11, provide functions to perform parallel computations on GPU which is benefitial for selected algorithm. The disadvantage of DirectX is that it is strictly bound to Microsoft Windows.

Other possibility is CUDA platform [CUDA13]. CUDA is parallel computing platform created by NVIDIA. It allows to write the parallel programs/kernels in C/C++ language. In CUDA, a programmer has better access to the memory of the GPU. A kernel can use both the global memory and the shared memory, which is shared amongst the threads in a block of the threads. The disadvantage of the CUDA is that it is supported only by NVIDIA graphics cards. Some of the steps of VCT rely on the rasterization of the scene. Although it is possible to program the rasterization on the CUDA platform, the special libraries such are DirectX and OpenGL or are the better choice. The CUDA is the possibility only for the non-rasterization steps of the VCT.

For our implementation we have chosen the OpenGL, due to its capabilities and multi-platformness. To produce multi-platform implementation, we have chosen the C++ programming language. C++ offers great performance, which is essential in real-time computer graphics. Many of the libraries, related to realtime computer graphics, are written in C or C++ and it is possible to use them without any additional wrapper.

The OpenGL is used to render the image into the framebuffer of the window. The creation and handling of this window is platform specific and is not provided by OpenGL. GLFW [GLFW13] and GLUT [GLUT13] are one of the libraries commonly used for this task. We have chosen the GLFW library. It provides provide all the necessary mechanisms for creating the OpenGL context, managing windows, user input, loading TGA images, timing and other support functions.

There are many formats to store 3D models, as well as there are many libraries to load and handle these different formats. One of the most widely used format for interchange of 3D models is *obj* format. We have chosen to use this format, because it easy to load, it supports material definition, many testing scenes are available in this format and there are many libraries capable of reading it. We have considered 3 loading libraries: kixor [KIX12], libObj [LOBJ13]and Assimp [ASS13]. For our implementatin we have choosed the Assimp library. It is capable of loading more then 30 formats and provide several functions for preprocessing the mesh.

5. Implementation

In this chapter we provide the description of our implementation. First, in the section Application structure 5.1, the general architecture of the program and rendering pipeline is shown. In the following sections, we describe each part of the rendering pipeline in detail. Please note that many illustrations show the 2D simplification of explained topic for better understanding.

5.1 Application structure

We have implemented Voxel Cone Tracing for Indirect Illumination using GLSL [GLSL] and C++ programming language. The implementation is built on OpenGL version 4.2 [OGL4]. OpenGL 4.2 features, including atomic counters and imageStore/imageLoad functions, are essential for our implementation. For the window handling and the OpenGL integration we have used the GLFW library [GLFW13]. The OpenGL extensions are handled using the GLEW library [GLEW13]. Matrix and vector operations, as well as related mathematics functions, are computed using the GLM library [GLM13], and the Boost library [Boo13] is used for various support functions, like type conversions or timers. The scene loading is handled using the ASSIMP library. All of the used libraries are cross-platform and the versions used in our implementation are shown in the table 5.1.

Library	Version
OpenGL	4.2
ASSIMP	3.0.1270
GLEW	1.9.0
GLFW	2.7.6
GLM	0.9.3.4
Boost	1.5.2

Table 5.1: List of the libraries and their versions used in our implementation

Application source is divided into several classes covering scene management and shader programs management, texture management, user interface and the rendering.

The application structure is adapted to the rendering pipeline of the voxel cone tracing. In the figure 5.1 we can see this pipeline. The sections of this pipeline are divided into several classes.

The scene is rasterized for the first time during octree generation process. During this rasterization a buffer of voxel-fragments is created. The voxel-fragments stored in this buffer are used for the octree generation. The functions and structures handling the voxel octree are contained in the class called *VoxelizationPass*.

Next, the scene is rasterized from the position of each light. Depth buffers created from the light serve as a visibility information for direct illumination. Light values are written to the the voxels lit by the lights. The functions and structures handling the direct illumination are contained in the class called *LightPass*.

The algorithm is designed for use with deferred renderer. In the last section of the rendering pipeline, the scene is rasterized from the view of a camera and the geometry buffer is stored. This geometry buffer is created using the class called *ExtractComponentPass*.

The geometry buffer produced by *ExtractComponentPass* is used by a class called *VCTPass*. This class handles actual Voxel Cone Tracing. For each pixel, visible from a camera, indirect illumination, specular highlights and ambient occlusion are computed. The results are stored in another buffers. The class called *MergeComponentPass* is handling the composition of final image. It uses buffers generated from *ExtractComponentPass*, *LightPass* and *VCTPass* and assembles them into the final image.

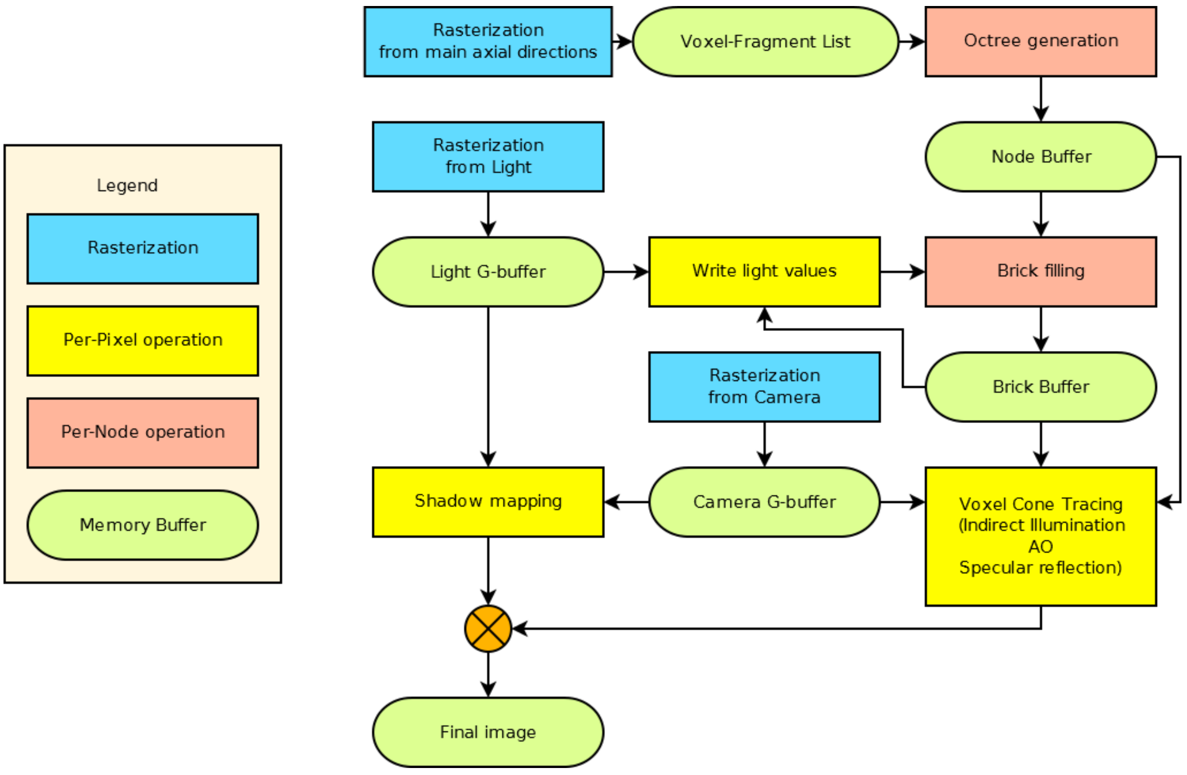


Figure 5.1: Rendering pipeline of our implementation

5.2 Voxel-Octree building

The octree structure, used for the rendering of indirect illumination and described in the chapter 2, is composed from the static and the dynamic part. The following description is focused on the static part of the octree. The differences for the dynamic part are discussed in the section 5.7.

The octree is used as a simplified representation of the scene. In order to get this representation most accurate, with limited depth, the bounding box of this octree must tightly encase the scene. In our implementation, we rely on cube-shaped voxels. In order to archive this, the octree bounding box is scaled, so in all dimensions, the size of the octree is the same.

The construction of the octree consists of several passes, which are shown in figure 5.2.

5.2.1 Voxel-fragments generation

The octree structure is built from the scene before the rendering phase. The octree is build by inserting the *voxel-fragments*, which are generated in *voxel generation pass*. In order to obtain these voxel-fragments

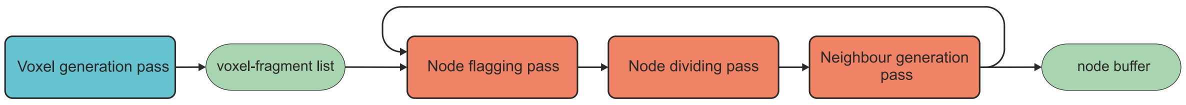


Figure 5.2: Octree generation process

from the scene, we render the scene with a orthographic projection. The orthographic volume of the projection is set to the size of the octree. The scene is projected into a framebuffer with resolution $2^d \times 2^d$, where d stands for the voxelization depth.

To get 3D distribution of the voxel-fragments, this projection must be performed in the 3 main axes. In our implementation the scene is projected only once, but each triangle is rotated according its normal in geometry shader. The geometry shaders have access to whole primitive, which is in this case a triangle. The normal of the triangle is determined from the position of the triangle's vertices. If the y or x components of the normal are dominant, the triangle is rotated by 90° in y-axis or x-axis.

The choose of the resolution, rotation matrix and projection volume causes, that the 2 of 3 coordinates in fragment's position, generated during rasterization process, correspond with the centres of the octree nodes in depth d.

After the rasterization, the inputs for the fragment shader are the fragment's position, normal and texture coordinates. In order to get the voxel-fragment's color, we perform a look up into the diffuse texture. The color must represent the color of all surface covered in a future octree voxel of depth d. To acquire such color, the lookup is performed using the function *textureLod* [OGL4], which allows to specify the mip-map level of the texture. This level is chosen uniformly for all objects in the scene and is one of the input parameters of the application. All the above mentioned attributes are inserted into *voxel-fragment list* for each fragment processed in the fragment shader.

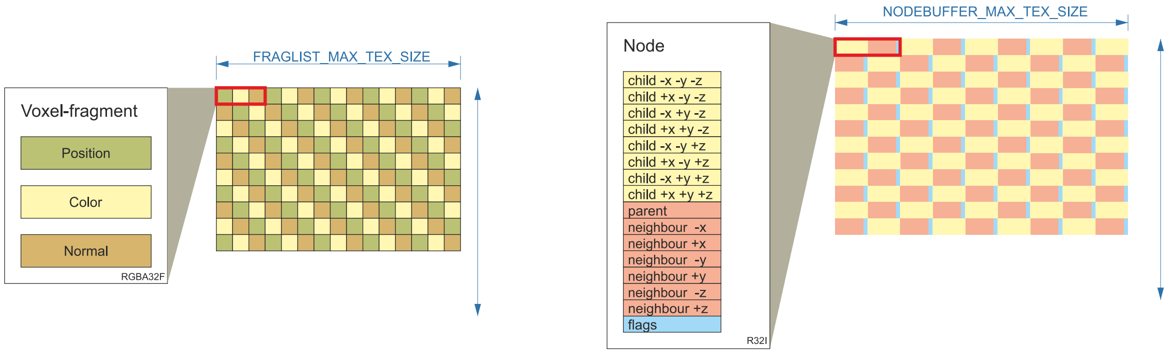


Figure 5.3: Both voxel-fragments (on the left) and octree nodes (on the right) are stored in 2D textures.

This list is composed of a buffer and an index pointing to the next free space in the buffer. The index is implemented using a *atomic counter*. The atomic counters were included in the core specifications in OpenGL version 4.2 [OGL4]. These counters presents a way of accessing, incrementing or swapping a variable in a coherent manner by many shader invocations. The buffer is, in fact, a texture. Each graphics card has a limited resolution of textures. The number of generated voxel-fragments is $< 2^{d^3}$ and, in most cases, higher than this limit for 1D texture. Table 5.2.1 shows these limits for 3 graphics card used for the testing. In our implementation we store the attributes of voxel-fragments in 2D texture and 1-dimensional index is mapped into 2-dimensional space. The code sample 5.1 shows how the single

```
//snippet from voxelization_genFragments.frag
layout(binding = 0, offset = 0) uniform atomic_uint fragID;
coherent uniform layout(binding = 0, rgba16f) image2D fragList;

in vec4 oPos;

ivec2 get2Dcoords(uint index){
   ivec2 coords;
   coords.x = int(index)%MAX_TEX_SIZE;
   coords.y = int(index)/MAX_TEX_SIZE;
   return coords;
}

void main()
{
   uint index = atomicCounterIncrement(fragID);
   ivec2 posIndex = get2Dcoords(index*VOXFRAG_STRIDE+FRAG_VALUEINDEX_POS);
   imageStore(fragList,posIndex,oPos);
}
```

Listing 5.1: The index of the voxel-fragment is obtained from an atomic counter. Values are saved in image2D.

value of voxel fragment is stored. For accessing the texture we use *imageStore* and *imageLoad* functions. These functions are operating on special data types image1D, image2D and image3D. These types are encapsulating data stored in one of the supported layouts. The layout used for the voxel-fragment list is *rgba32f*.

Attribute	GeForce GTX 660	GeForce 550M	GeForce GTX 580
GL MAX TEXTURE SIZE	16384	16384	16384
GL MAX 3D TEXTURE SIZE	2048	2048	2048

Table 5.2: Maximal size of textures allowed by graphic cards

After the projection, each fragment generated in the voxel generation pass presents one voxel-fragment with position, normal and color. The visualisation of such voxel-fragments is shown in figure 5.4.

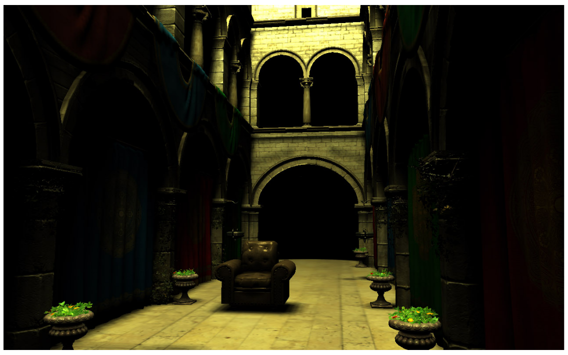
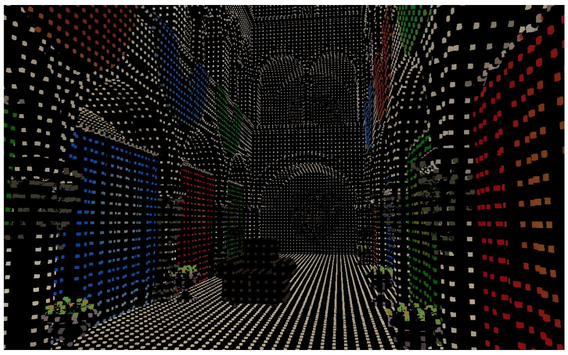



Figure 5.4: The Sponza scene rendered with direct illumination (on the left) and with the visualization of the generated voxel-fragments (on the right)

5.2.2 Octree structure

In our implementation we have chosen an approach where each node consist of 16 values. A node contains 8 pointers to a child node or leaf, a pointer to the parent of the node, 6 pointers to the neighbours of the node and a flag. All of these values are stored as 32-bit integer. The GLSL language does not support pointers to a memory. Instead we use the offset from the beginning of a buffer. We call these offsets the pointers in further text. Using this structure we do store only non-empty nodes. Another advantage of this approach is that leafs of the octree can point to a voxel-fragment in the voxel-fragment list. This means we do not have to create nodes for the last level of the octree.

All the nodes are stored in a *node buffer*. In our implementation this buffer is created as the 2D texture with the *R32I* format as shown in the figure 5.3. The number of the nodes is not known before the actual octree construction is finished. Therefore, the texture resolution of the node buffer is set by the parameter NODEBUFFER_ MAX_ SIZE. The node buffer values are pre-set to the value -1 since the value 0 is used for the pointer to the root node. The top of the buffer is implemented by another atomic counter.

5.2.3 Octree construction

For the octree construction we have decided to implement the algorithm described by Cyril Crassin and Simon Green from 2012[CG12]. The advantage of this algorithm, over the one mentioned in article *Indirect Illumination using Voxel Cone Tracing* [Cra+11] is, that the octree is constructed in a level by level way. This ensures, that the nodes on the same level of the octree are located consecutively in video memory. The advantages of this ordering are discussed in section 5.4

The algorithm, we have implemented, builds the octree iteratively from the top. Each iteration consist of three passes, shown in figure 5.2. All these passes are performed by executing shader programs on known number of elements - nodes and voxel-fragments.

To run a specific count of threads *n* in OpenGL 4.2, we draw *n* vertices. The rasterization is disabled as we rely only on vertex shaders. In the GLSL, the in-build variable *glVertexID* serves as a identifier for the thread. For actual drawing, we use function *glDrawArrays*, which is used with an empty vertex array object. The element count is set to the number of the threads.

In the first pass, we execute a thread for each fragment in the voxel-fragment list. The fragment is traversed through the octree, which is initially only the root node. The function for determining the position of voxel fragment in the octree is given in code sample 5.2. Once the deepest level of the current iteration's octree is reached, the leaf is flagged. To flag a leaf, we set it's value to 1.

In the second pass, all the flagged nodes are subdivided. We execute a thread for each leaf of the currently build octree. If the leaf is flagged, the shader program obtain the address of the free space in the node buffer from the atomic counter. The leaf is set to point on the newly created node. This two-step approach is chosen to prevents conflicts, which would appear if one leaf is being subdivided by two or more fragments.

The third pass of the octree construction is used to determine the neighbours of the nodes. We have implemented this process in separate pass to prevent conflicts. All the leaves flagged during an iteration must be divided before the neighbours are determined.

In the last iteration, only the first step is performed. The leaf is not flagged this time, instead, we store the pointer to the voxel-fragment, which lies within the leaf. After *d* iterations the octree with depth *d* is built. The examples of the approximation of the scene by the octrees with various depth is shown in figure 5.5.

```
//snippet from voxelization_flagNodes.vert
int traverseOctree(vec3 pos){
   vec3 minBB = sceneMinBB;
   vec3 maxBB = sceneMaxBB;
   int nodeIndex = 0;
   int nodeBase = 0;
   for(int d=0;d<MAX_DEPTH;++d){
      vec3 halfSize = (maxBB-minBB)/2;
      vec3 relativePosition = pos - (minBB+halfSize);
      vec3 mask = clamp(sign(relativePosition),0,1);
      int offset = (int(mask.z)<<2)+(int(mask.y)<<1)+int(mask.x);
      maxBB = minBB+halfSize+(halfSize*mask);
      minBB = minBB+(halfSize*mask);
      nodeIndex=nodeBase+offset;
      nodeBase = imageLoad(nodeBuffer,get2Dcoords(nodeIndex,NODEBUFFER_MAX_SIZE)).r;
   }
   return nodeIndex;
}
```

Listing 5.2: Traversation function used in node flagging pass

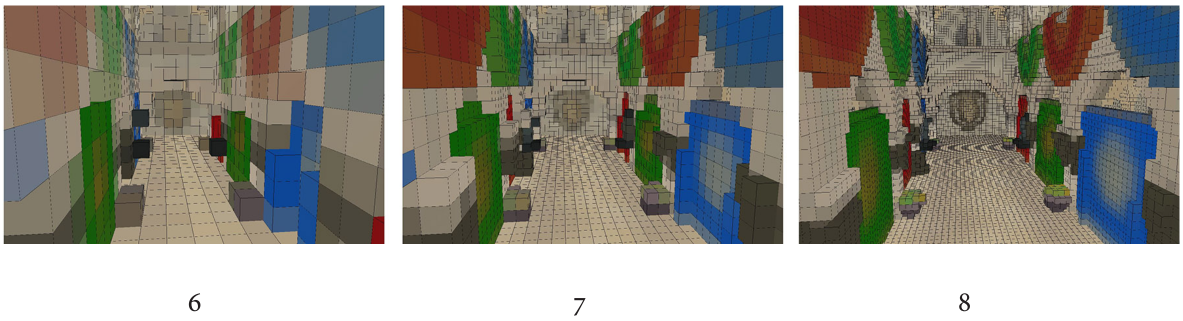


Figure 5.5: The octree visualization; the depth of the octree, from left to right, is: 6, 7, 8

5.3 Voxel representation

Our implementation supports 2 models of voxel's light information.

The first model stores the outgoing radiant intensity for each one of the six main axes. The voxel is represented by 8 variables. These variables are color, opacity and outgoing radiant intensity in the 6 axes. All these variables are stored as a 3D vector.

The second model stores the incoming radiant intensity to the voxel and the direction to the light source. The variables stored for this model are color, opacity, normal of the surface, direction to the light and incoming intensity. Normal and the direction to the light are interpreted as gaussian lobes. The shading of the voxel is computed from these variables.

If we store only one value per variable, the discontinuity of the surface parameters will negatively affect the rendering quality. In order to improve this quality a value must be interpolated for each sample from adjacent nodes. This method utilizes hardware tri-linear interpolation to get smooth transition between the values. To allow this interpolation, each variables of the voxel is represented by a brick. The brick is a matrix of 3^3 cells as is shown in the figure 5.6. These cells are not located in the middle of the node children's area, but they are located on the edges of the children nodes. The cells on the edge of the brick are duplicated in the adjacent nodes. The bricks are stored in a 3D texture and each brick has a block of 3^3 values in this texture. The difference between this representation and the one, where the variable is stored as single value is shown in the figure 5.7.

The bricks are stored in a brick buffer, which is a 3D texture with a compact OpenGL specific format

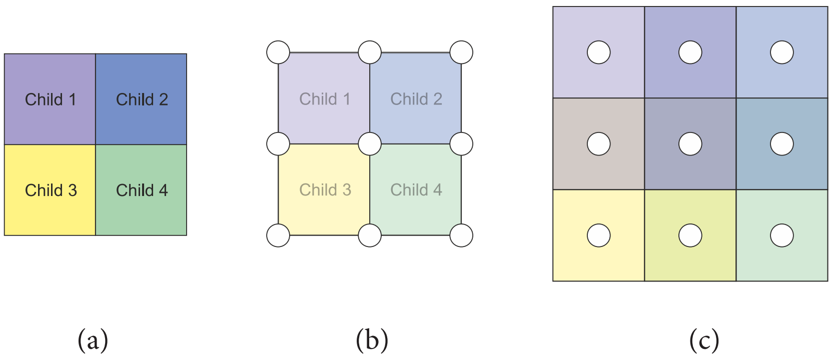


Figure 5.6: The node and its children (a); brick values are located in the corners of node's children (b); a brick is represented by 3^3 values (or 3^2 values for 2D case) (c)

r11f_ g11f_ b10f [OGL4]. This format stores 3-channel color in 32bits, and usage of this format significantly reduce the memory consumption of the brick buffer. This format doesn't support negative values and therefore normalized vectors must be transferred to range $[0, 1]$.

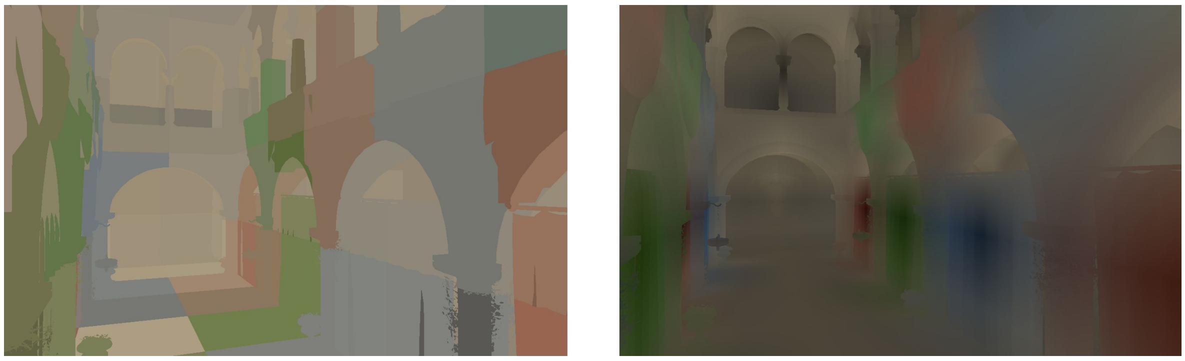


Figure 5.7: The color of the voxel mapped to the surface of the geometry; the color is represented by single value per voxel (on the left); the color of the voxel is represented by brick of 3^3 values (on the right)

5.4 Brick filling

Some of the voxel's parameters are acquired during the light pass. Other parameters do not change, and are acquired after the voxel octree is build. These static parameters are color, normal and opacity.

In order to fill all the bricks with the right values, we perform several steps. Initially, all the brick's values are set to (0,0,0). Next, the bricks of the deepest level of the octree must be filled with the values acquired from the voxel-fragments. Then, iteratively from the deepest level of the octree up to the root, the values are acquired from the children bricks and the edge values are transferred between neighbours.

All these steps are implemented as a separated shader programs and these programs are executed on a specific number of bricks. In order to run a specific number of threads we use the same approach as was used for passes involved in the octree construction 5.2.3.

As mentioned in section 5.2.3, the nodes of the same level are located consequently in memory. The

```
//pseudocode - fillings of the brick
void fillBricks(){
   clearBrickValues();
   gatherValuesFromVoxFragments();
   copyValuesInside();
   copyToNeighbours(VOXELIZATION_DEPTH);
   for(int vd=VOXELIZATION_DEPTH-1;vd>0;--vd){
      gatherValuesFromChildren(vd);
      copyToNeighbours(vd);
   }
}
```

Listing 5.3: Pseudocode of the brick filling process

same applies to the bricks. The brick index corresponds with the node index. This memory arrangement allows us to access the bricks of specific level easily, without the need for traversing the octree.

5.4.1 Gathering values from voxel fragments

The last level of the octree is composed by the pointers to voxel-fragments. The location of a pointer determine one of 8 overlapping sections of the brick, which is to be filled with the voxel fragment values. The bricks are filled with these values in 2 passes.

In the first pass of the process, a value of the voxel fragment, is stored in left front top most position in the appropriate section of the brick.

The second pass is used to spread the initial values inside the brick. First, the values inside the brick are added to the values on next position in the positive x-axis direction. The same process is repeated for y-axis and z-axis. The two passes, used to fill the bricks with voxel fragments values, are illustrated in the figure 5.8.

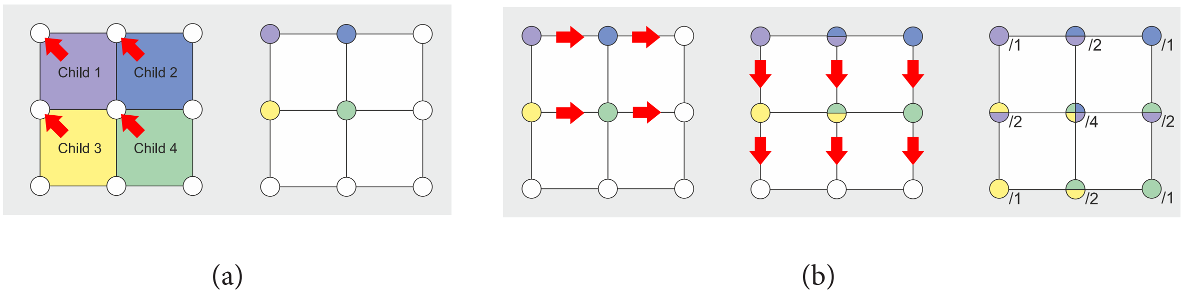


Figure 5.8: The value of the voxel fragment is stored in left top most position in the brick (a), then the values are copied in main axial directions inside the brick and averaged (b).

5.4.2 Brick edge copy

A brick's location in a memory does not correspond with the location of the voxel in a space. Two adjacent voxels are generally not adjacent in the memory. To overcome this fact and to get continuity between two voxels, the edge values of the bricks are duplicated in two bricks of adjacent voxels. Generally, after the gathering of the values from either voxel fragment or children's bricks, the edge values of two adjacent voxels may differ. In order to get smooth transition over the voxels, these edge values must be averaged.

This edge averaging is performed in a new pass. In order to average the edge values in x-axis direction, for each node, we add the edge values of the neighbour brick on negative-x side to appropriate values in the current node. These values are divided by 2, to get the average. In the second step, the edge values from neighbour brick on positive-x side, which now contains the average computed from a different thread, are just copied. The illustration of this process is shown in figure 5.9. The very same process is repeated for y-axis and z-axis.

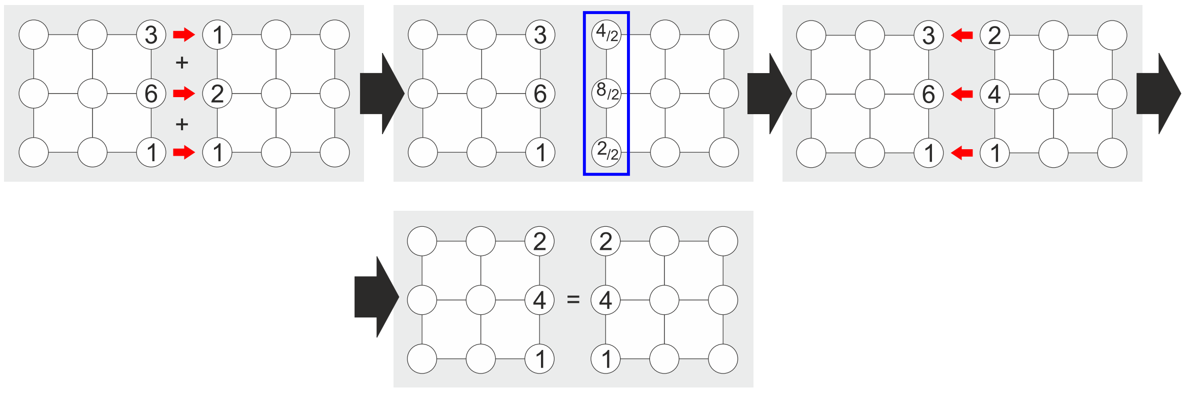


Figure 5.9: The illustration of the brick edge copy process

Once we use this process on the sparse voxel octree, continuity errors appears. The example of origin of these errors is given in figure 5.10.

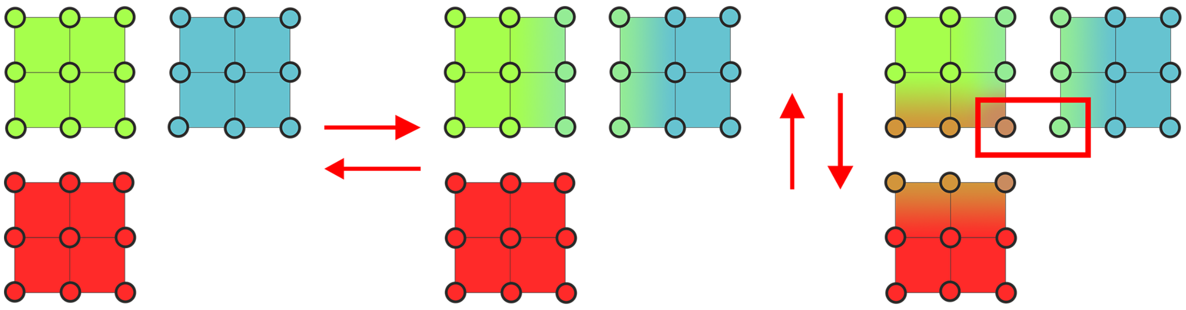


Figure 5.10: If one of the neighbour is missing, the discontinuity, marked in the image by red rectangle, occurs.

5.4.3 Brick averaging

Once a level of the octree is filled with the correct values, we use this values to compute content of the bricks from the higher level of the voxel octree.

One value of a higher level node's brick is computed as a average of 3^3 values, acquired from the 8 bricks of the node's children. These 8 brick may not be direct children of the higher level brick, as is shown in figure 5.11. The node have access only to it's children. To overcome this limited access, the values of children's bricks are used to compute incomplete average of the brick values. According to the value's position in the brick, from 8 up to 27 values must be acquired from the children to compute this incomplete average. The process is then completed by the edge copy pass described in the section 5.4.2.

In our implementation we execute a thread for each of 3^3 brick's values. The shader program is common for all the values in a brick and according to the position of a brick value, it gathers 0 or 8

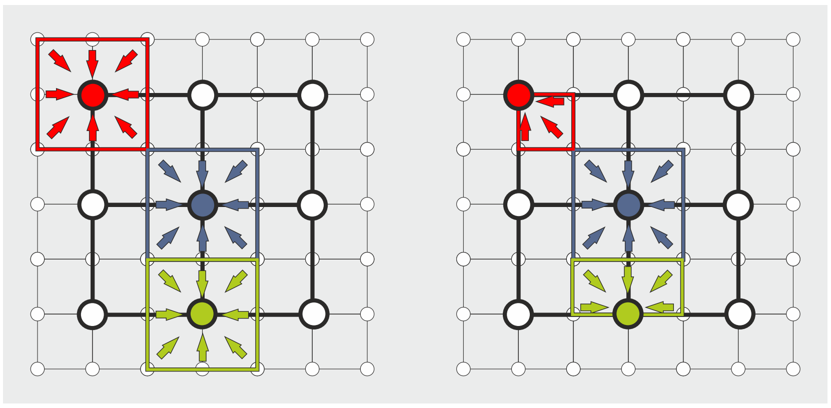


Figure 5.11: The correct value of higher level brick is average of the 3^3 values from deeper level bricks (on the left). The actual value is averaged only from the values of the node's children (on the right).

values from each of the children bricks. The final averaged value is computed as weighted average, where the opacity of the value is used as the weight.

In our implementation we store the opacity as a 3D vector. The opacity represent the percentage of blocked light in one of 3 main axial directions. The illustration of directional opacity is shown in figure 5.12.

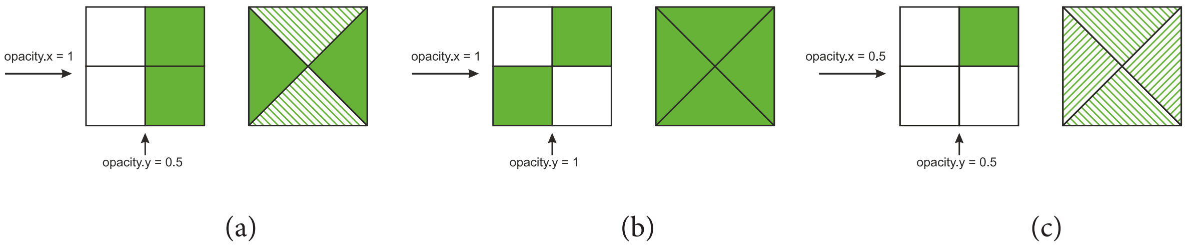


Figure 5.12: The opacity is represented directionally.

For the opacity value, the brick averaging pass is modified. When we are computing the opacity, we first compute the maximum opacity in 3 directions for all these values. The higher brick value is then averaged from 3^2 values of maximums for each direction, as is illustrated in figure 5.13.

5.5 Light injection

In our implementation, all the lights are omnidirectional. For each light we store it's position and intensity. The direct illumination is implemented using the shadow mapping technique [Wil78]. To render a shadow map, a scene is rasterized from the position of a light. We store the depth of each fragment, generated during this rasterization. The shadow maps are stored in a texture array. A single texture from this array has resolution determined by a parameter, chosen by a user. It is one of the performance relevant parameters of the implementation. For purpose of the shadow map rendering, the omnidirectional light is divided into 6 orthogonal frustums with the field of view of 90°. Each frustum of the light is rendered into it's own texture layer.

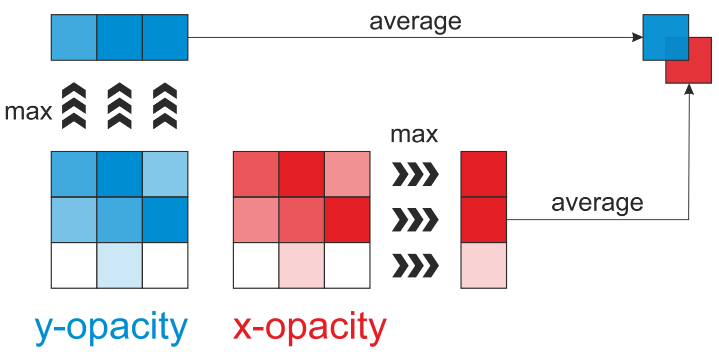


Figure 5.13: In each direction the maximum opacity values are obtained from children bricks, the opacity value is then composed from the average of these maximums.

To fill the voxel octree with the light information, we perform several steps. First, in a new pass, we convert the depth information of each texel of the shadow map into the index of the brick and the section of the brick. To achieve that, the texel position, computed from the depth of the texel, is traversed through the octree using a similar function to one shown in code sample 5.2. The relative position acquired during this traversation is used to determine one of the 8 subsections of the brick.

The traversation function, used in this pass, also flags all the nodes, which are visited. This flag is used to create an update list, which is used to reduce the number of filtered nodes and is explained further in this section. Both the brick index and the brick section are integer variables and are stored in 2 buffers. These buffers are 2D textures of format R32I. The resolution off these buffers is the same as the resolution of the shadow map.

The brick index and section determine the location in the brick buffer, where the light information is to be written. In another pass, for each texel, the shader program checks the value of brick index and the section of the brick processed by adjacent (in the meaning of the *gl _ FragCoord* value) threads. If two or more threads are processing the same section of the same brick, only the top left most thread continues. The other threads are terminated. This solution was chosen to reduce the amount of concurrent memory accesses.

As described in the section 5.3, our implementation supports 2 representations of the voxel light. The first model stores the radiant intensity outgoing from the voxel. This intensity is stored for 6 main axes. The intensity for an axis is computed as $\textit{light intesity} \times \textit{voxel color} \times \textit{quadratic attenuation} \times \cos\theta$. Where the θ is the angle between the light direction and the axis.

The second model stores incoming radiant intensity and the direction to the light source, expressed as gaussian lobe. The direction to the light source is stored as the normalized vector. The incoming radiant intensity is modulated by quadratic attenuation.

After each of the shadow maps is processed, the bricks with the light related informations are filtered to the upper levels of the octree. Filtering is done the same way as the filtering of the color and normal parameters of the voxel, which was explained in the section 5.4.

The implementation supports only one light to be involved in the indirect illumination. Multiple lights would require atomic operations on float numbers or the use of a mutex. The float atomic operation requires *NV_shader_atomic_float* OpenGL extension, which is available on modern nVidia graphics cards.

Once the light information is written in the last level of the octree, this information is filtered into the higher levels. Naive solution is to filter all the nodes. In our implementation we reduce the number of the filtered nodes by using a *update list*. The update list contains indices of the nodes that needs to be filtered. These indices are stored in a level by level manner, starting from the deepest level and ending with the root node. The update list is composed of the buffer, which is a 2D texture with format R32I, and an atomic counter pointing to the next position in the update list.

In the light injection process, all the nodes that were visited in the traversation function were flagged. The node is flagged if the integer located in the 16th position of the node is set to 1. To create the update list, we run a shader program for each level of a voxel octree, starting with the deepest level. In each iteration, we run the program on all nodes in the voxel octree level. The shader program checks the flag. If the flag is set to 1, a location of a free space in update list is acquired from the atomic counter. The index of the node is stored in this location. The flag is then reset to its default value.

All the passes used for brick filtering are modified to work with this update list. The thread index is not used to point into the node in memory, but instead it is used point into the update list. The actual node index used for filtering is then acquired from the update list.

5.6 Voxel Cone Tracing

The actual cone tracing is performed once all the bricks of the voxel octree are filtered. We have implemented the voxel cone tracing in a separate pass. The pass writes into two render targets. The first render target is a texture buffer of RGBA colors, which stores the diffuse part of indirect illumination and ambient occlusion. The second render target is a texture buffer of RGB colors, which stores the specular part of indirect illumination.

For each visible fragment, the shader program reads the depth and the normal of a texel. The light is gathered from several cones across a hemisphere. In our implementation a cone placement simulates the Phong shading model [Pho75]. The number of these cones is determined by the user. The direction and the span of each diffuse cone is precomputed on CPU in accordance with the number of the cones. The precomputed cone directions are rotated by the normal of the texel and the cones origins are set to world-space position of the texel, which is computed from the depth. The specular cone, which is used for glossy surfaces, has tight span, which is derived from the specular exponent of the material . The illustration of the cone placement is shown in the figure 5.14.

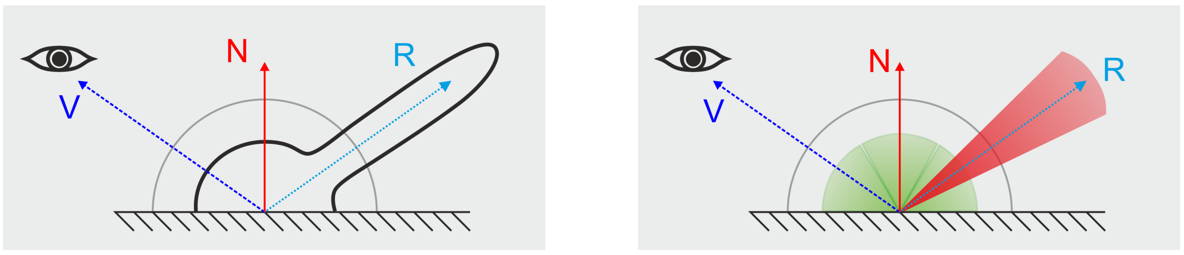


Figure 5.14: The illustration of the cone placement in 2D; The Phong shading model(on the left) is approximated with several wide diffuse cones and one tight specular cone (on the right).

To effectively gather the intensity coming from a cone, we generate several samples along the direction of the cone. The illustration of the sample placement is given in the figure 5.15.

For each such sample we perform a lookup into the voxel octree. The distance l from the cone origin and the cone span α determine the sample diameter $r = 2l \tan\left(\frac{\alpha}{2}\right)$. The depth d of the octree, in which

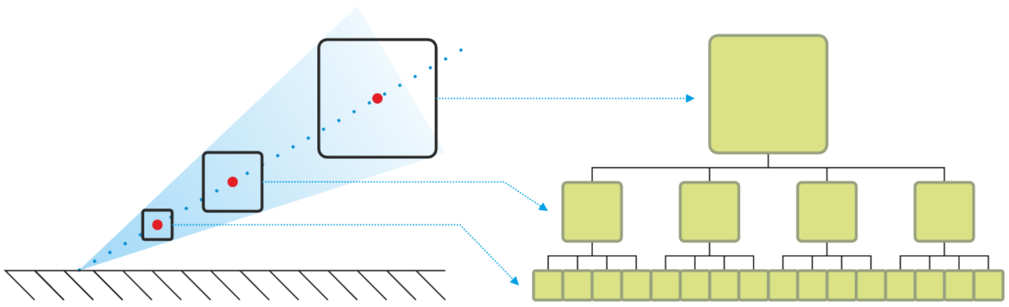


Figure 5.15: The diameter of the cone at the sample position is equal to the depth of the lookup to the octree (the 2D case with a quadtree).

the lookup is made, corresponds to the sample diameter r and is expressed by equation 5.1, where w_s denotes the width of the bounding box of the octree.

$$d = \log_2\left(\frac{w_s}{r}\right) \tag{5.1}$$

In our implementation we use two different sampling methods. The first method is used for diffuse indirect illumination, which is gathered using several wide cones. We generate sample positions, in a way, that each sample diameter correspond precisely with the size of a voxel in a particular level of the voxel octree. The first sample is taken from the deepest level of the octree and each subsequent sample is taken from the higher level of the octree. This sampling method is fast and is suitable for wide cones. For tight cones, this sampling method generates the samples to far from each other and thus the possibility of completely missing the light obstacle emerge. Tight cones, which are used for rendering glossy materials, are sampled in a different manner. The distance from a surface point to the first sample is chosen in a way that the sample diameter is equal to the size of a voxel in the lowest level of the octree. Each subsequent sample is placed right next to the previous sample. The distance between two successive samples is equal to the sample diameter of the prior sample.

Each sample is traversed through the voxel octree, until the desired depth is reached. The traversation function return not only the index of the node, but also the relative position inside the node. If the depth is not integer value, the traversation function is modified to return a pair of the nodes' indices and relative positions.

The node index and the relative position are used to acquire values from the brick buffer, which is as attached to the shader program as *sampler3D*. Every lookup, done by GLSL function *texture*, in the brick buffer results in tri-linearly interpolated value. This interpolation is done by hardware, without any additional instructions. If the lookup depth is not integer number, the values from are gathered from two voxels. These values are then linearly interpolated. Together with HW interpolation the single sample value is quadrilinearly interpolated.

For each sample we gather the radiant intensity coming from the voxel to the lighted surface point. As mentioned in the section 5.3 and the section 5.5, we have implemented two voxel light models. The first model is composed of six values of radian intensity coming to six main axes. To acquire the radiant intensity incoming from such voxel to a surface point, we sum the values of radiance from all six axes modulated by projected area. The GLSL function to acquire the intensity of the voxel is shown in code sample 5.4.

The second voxel light model is composed of color,radiant intensity,light direction and normal. The

```
vec4 getLightIntensity(vec3 direction,brickParams parameters,float slope){
        vec3 radiance = vec3(0);
        vec3 opacity = getTexValue(parameters,BRICK_VALUEINDEX_OPACITY).xyz;
        ivec3 component = clamp(ivec3(sign(direction)),0,1);
        vec3 portion = abs(direction);
        radiance+=getTexValue(parameters,BRICK_VALUEINDEX_RADIANTI_X_MINUS+component.x).xyz*portion.x;
        radiance+=getTexValue(parameters,BRICK_VALUEINDEX_RADIANTI_Y_MINUS+component.y).xyz*portion.y;
        radiance+=getTexValue(parameters,BRICK_VALUEINDEX_RADIANTI_Z_MINUS+component.z).xyz*portion.z;
        return vec4(radiance.rgb,max(dot(opacity,portion),0));
    }
```

Listing 5.4: The computation of the light intensity radiating from the voxel - the represatation using radiant intensity for 6 axes

```
vec4 getLightIntensity(vec3 direction,brickParams parameters,float slope){
        vec4 color = getTexValue(parameters,BRICK_VALUEINDEX_COLOR);
        vec4 normalLobe = gaussianLobeFrom01(getTexValue(parameters,BRICK_VALUEINDEX_NORMAL).xyz);
        vec4 reflectDirLobe = gaussianLobeFrom01(getTexValue(parameters,BRICK_VALUEINDEX_LIGHTDIR).xyz);
        reflectDirLobe.rgb = reflect(reflectDirLobe.rgb,normalize(normalLobe.rgb));
        vec4 incomingIntensity = getTexValue(parameters,BRICK_VALUEINDEX_LIGHTINTENSITY);
        vec3 opacity = getTexValue(parameters,BRICK_VALUEINDEX_OPACITY).xyz;
        vec4 coneLobe = vec4(-direction,cos(slope));
        float reflectanceProbabilityToPoint =
            gaussianFromLobe(convGaussianLobes(convGaussianLobes(normalLobe,reflectDirLobe),coneLobe),-direction);
        vec3 reflIntensity = color.rgb*incomingIntensity.rgb*reflectanceProbabilityToPoint;
        return vec4(reflIntensity,dot(abs(direction),opacity));
    }
```

Listing 5.5: The computation of the light intensity radiating from the voxel - the represatation using gaussian lobes

normal of the underlying surface and the light direction are expressed as gaussian lobes. We compute the gaussian lobe expressing the probability of emitting the light in a direction as convolution of normal, light direction and cone span lobe. By evaluation of that gaussian lobe in the direction to the surface point, we get the amount of reflected intensity. The radiant intensity coming out of the voxel to the surface point is the voxel's intensity multiplied with the calculated amount of reflected intensity. The GLSL function to acquire the intensity of the voxel from the gaussian lobe representation is shown in code sample 5.5.

The acquired values of incoming radiant intensity from the samples are accumulated by the equation 5.2, where I_{prev} denotes the accumulated intesity, a_{prev} denotes so far accumulated opacities and I_{sample} denotes the intensity of a sample. The opacities of the samples are summed, and the tracing of a single cone is finished once the accumulated opacity is greater or equal to 1.

$$I = I_{prev} + \left(1 - a_{prev}\right) * I_{sample} \tag{5.2}$$

5.7 Dynamic scenes

The sparse voxel octree, representing the scene, is build once and its building is the part of the preprocessing. In order to enable the dynamic objects to affect the indirect illumination, it is necessary to integrate these dynamic objects to the voxel octree.

Naive solution, which is to rebuild the octree each time the dynamic objects change, is not effective, as can be seen in measurements in table 6.11. Instead, we divide the octree to a static and a dynamic part. In our implementation the nodes and the bricks of the dynamic part are inserted into the memory after the static part. Every time, the update is made, the whole dynamic part of the octree is deleted, and new voxelization is performed. To fill the brick buffer part of newly created nodes, we rely on the update list mechanism described in section 5.5. Since the bounding box of the octree is derived from the static part of the scene, only dynamic objects located inside the static part are supported.

To delete the dynamic part, it is necessary to reset the nodes located in the dynamic part of the memory to it's default unfilled state and to reset the references from the static part to the dynamic part. This is done by GPU by another pass.

After the node buffer contains only the static part, the atomic counter pointing to the top of the voxel-fragment list is set to the count of the static voxel fragments, and the atomic counter pointing to the top of the node buffer is set to the count of the static nodes. The dynamic part is voxelized using the same 3 steps described in section 5.2.3, but with little modifications. The pass responsible for flagging the node for the division also flags the node to be updated. The neighbour generation pass is also modified. Once the neighbour of a dynamic node are computed, the pointer to this dynamic node is written to appropriate slot in the neighbours' nodes. These adjacent nodes are also flagged to be updated.

The brick buffer's size is set during the voxelization of the static part. In that moment, the amount of dynamic nodes is not known. The size of the brick buffer is enlarged by a factor, accessible as command line parameter. The default value is set to 20 % and this value is sufficient to all the scenes tested in chapter 6.

To update the brick buffer it is necessary to fill not only the newly created nodes, but also the adjacent nodes. All the nodes, which needs to be refilled with values, are flagged after the octree building process. In the next step, the flagged nodes are inserted into the update list. This update list is then used to determine nodes for brick filtering process.

6. Results

In this chapter we provide both performance and quality analysis of our VCT implementation. In the section 6.1, we describe the scenes we have used for the testing. The section 6.2 contain the description of used hardware setups, as well as comprehensive measurement of all the aspects of our implementation. In the section 6.3 we show the impact of the various setting to the quality of the rendering. We also compare the results with Imperfect Shadow Maps in this section.

6.1 Tested scenes

Our implementation was tested on 5 scenes. The scenes and their attributes are listed in tables 6.1 and 6.2. The four of the selected scenes were often used for testing of global illumination algorithms, and presents various levels of the geometry and light transport complexity.

The first of the scenes used for the tests is called Cornell-box. It's well-known and frequently used synthetic scene. The main element of the scene is a box, with one open side. The walls of the box are coloured in different colors. The main purpose of this scene to show the color bleeding effect.

Second scene, we have used for testing, is Crytec Sponza scene. The scene was originally modelled by Marko Dabrovic and the inspiration for the scene was the Atrium Sponza Palace in Dubrovnik. Crytec remodelled the scene and added new elements like vases, foliage and color curtains. Also the materials were improved by using normal maps. Sponza scene is middle sized scene with medium geometry and high ligth transport complexity.

Third scene, the Conference room, is small scene with high geometric complexity. The scene is a simple single room, with lots of objects such as chairs and tables. The main focus for the testing of the indirect illumination is the space below the table. This space contains many objects, which block the light transfer.

As the fourth scene, we have used the model of Sibenik cathedral. Sibenik is spacious scene with simple geometry and materials. The architecture of the cathedral contain many columns and small walls, which leads to small shadowed areas.

The fifth scene, which is the only one we have modelled, is simple U-shaped corridor. The scene is designed to reveal light leaking effect of the algorithm. The scene contains some brightly coloured elements to show *color bleeding* effect and the floor with reflective material.

6.2 Performance analysis

Our implementation was tested on several computers. The specifications of these computers are listed in the table 6.3.

To properly test all the aspects of our implementation, we have added moving objects, moving light and moving camera to the scene. The duration of all the steps of VCT, tested on various HW, are listed in table 6.4. The static part of the octree is build and filled only once, other steps are performed each frame. With our implementation we were able to reach interactive frame rates with the GeForce GTX 660 and 580. Measurements shows, that the most expensive step of the algorithm is light injection step. It often takes more than 30 % of the total time spent on the rendering of single image. The measurement also clearly shows that the performance is not dependant on the complexity of the scene.

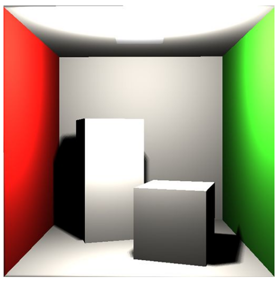
			
Num. of triangles	36	262267	331179
Textures	No	Yes	No
Geometry	Simple	Hard	Hard
Lighting	Simple	Hard	Medium

Table 6.1: Tested scenes with number of triangles, material definition, geometrical and lighting complexities

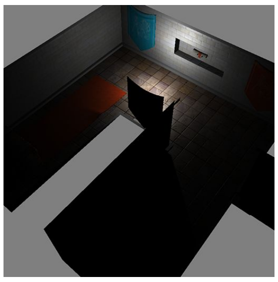
		
Num. of triangles	75282	5634
Textures	Yes	Yes
Geometry	Medium	Simple
Lighting	Medium	Hard

Table 6.2: Tested scenes with number of triangles, material definition, geometrical and lighting complexities

We have implemented 2 diffesnt representation of voxel light information. The respresentation using the gaussian lobes is using less brick and therefore the process of the bricks filtering is faster as is shown in table 6.5. On the other hand the evaluation of the single sample during cone tracing is slower. If we compare the total time that includes both light injection step and cone tracing then the represatation using gaussian lobes is slightly faster.

The resolution of the final image affects only the cone tracing step. As shown in table 6.7 the time needed for the cone tracing grows linearly with the number of final image pixels. The resolution of shadow map affects the initial steps of light injection. Since we use omnidirectional lights it is necessary to traverse 6 shadow maps for a light. With the increasing resolution of the shadow maps the time of traversation step grows immensely. In practical use it is too costly to use the shadow maps larger than 1024×1024.

The performance of all the steps of the algorithm are dependent on the octree depth as is shown in table 6.9. The lower the depth is, the less nodes have to be constructed and the less bricks are to be filled with values.

Another parameter of the VCT, which affect the performance is the number of cones used for diffuse

HW Configuration	A	B	C
Graphics card	GeForce GTX 660 2GB	GeForce GT 550M 1GB	GeForce GTX 580 1.53GB
Processor	Intel Core i7 3770K	Intel Core i5 2410M	Intel Core i7 2600K
Memory	16 GB DDR3	6 GB DDR3	16 GB DDR3
Operation system	MS Windows 7 64-bit	MS Windows 7 64-bit	MS Windows 7 64-bit
Driver version	314.22	314.07	314.22

Table 6.3: The specifications of computers used for testing.

HW conf.	Scene	Oct. build	Brick fill	Dyn. oct. build	Dyn. brick fill	L. inject.	VCT Diff	VCT Spec	Total time
A	Sponza	31.80	81.79	4.09	36.86	40.90	17.22	14.09	113.15
	Conference	21.51	48.72	3.96	46.18	55.95	16.05	7.78	129.92
	Sibenik	24.10	60.29	4.60	50.45	59.25	15.92	14.20	144.42
	Cornell box	18.76	51.49	4.92	70.61	77.75	9.87	7.43	170.59
	Dungeon	18.68	45.05	4.75	43.85	48.70	14.73	10.65	122.68
B	Sponza	146.10	263.56	8.36	118.86	201.90	86.03	64.48	479.63
	Conference	66.93	122.94	9.23	151.80	350.91	80.10	35.78	627.82
	Sibenik	83.63	186.80	8.83	164.22	354.82	81.61	64.23	673.71
	Cornell box	65.95	155.49	14.88	234.73	387.60	52.76	37.79	727.76
	Dungeon	63.57	129.71	19.68	133.52	326.52	71.77	49.21	600.70
C	Sponza	41.05	43.76	2.15	22.89	34.37	32.79	9.81	102.01
	Conference	18.08	21.26	2.35	27.72	53.77	30.47	5.73	120.04
	Sibenik	24.97	31.26	2.09	30.13	55.37	30.01	9.80	127.40
	Cornell box	17.81	26.93	3.89	42.47	61.38	18.75	5.97	132.46
	Dungeon	16.23	22.49	4.15	26.52	50.06	24.46	7.41	112.61

Table 6.4: Performance of the VCT steps tested on 3 HW setups 6.3; All times are in milliseconds.

indirect illumination. The table 6.10 shows the duration of cone tracing for various number of cones. With the increasing amount of cones, the duration of cone tracing grow linearly.

The construction and filling of the octree involves many passes. The table 6.11 shows the duration of these passes for the static part of the scene and the table 6.12 shows the duration of these passes for the dynamic part of the scene. The main bottleneck of the filling process is the pass, where the brick values are gathered from children. This pass involves many memory accesses and instructions.

The light injection measurement results are shown in the table 6.13. The voxels lit by a light source must be filled by same filtering process, which is used for initial brick filling. The gathering values from children last the most time here as well. The amount of time spent on filtering is dependent on the number of the voxels lit by the light. Update list generation on the other hand is dependent on the number of all the nodes of the octree.

The majority of the VCT memory requirements are taken by the brick buffer. The memory re-

Voxel representation	Scene	Oct. build	Brick fill	Dyn. oct. build	Dyn. brick fill	L. inject.	VCT Diff	VCT Spec	Total time
6 axial radiant intensity	Sponza	28.4524	82.22	2.60	34.60	41.05	16.87	14.05	109.18
	Conference	13.35866	39.31	2.42	44.26	54.07	16.01	7.49	124.25
	Sibenik	18.12558	59.09	2.48	48.37	57.60	16.04	14.00	138.49
	Cornell box	12.7242	51.16	3.07	67.49	75.85	10.63	7.92	164.96
	Dungeon	11.8201	41.86	3.00	42.31	49.38	13.69	10.38	118.77
gauss. Lobes	Sponza	34.1763	107.10	4.58	23.74	26.14	20.72	17.51	92.70
	Conference	18.54325	48.20	4.18	25.14	34.58	18.57	8.08	90.55
	Sibenik	23.03497	71.82	4.18	26.70	34.48	17.99	16.20	99.54
	Cornell box	17.42828	63.35	4.65	35.73	42.04	11.56	8.78	102.77
	Dungeon	16.87407	51.32	4.67	23.82	31.81	15.10	11.68	87.08

Table 6.5: Performance of the VCT with the representation using 6 intensity values and with the representation using gaussian lobes; All times are in milliseconds.

Vox. rep.	axial radiant intensity			gauss. Lobes		
Oct. depth	7	8	9	7	8	9
Sponza	24	108	431	24	72	276
Conference	7	60	204	11	36	132
Sibenik	24	84	323	10	48	204
Cornell box	24	84	323	24	60	204
Dungeon	24	60	216	8	36	132

Table 6.6: The memory requirements of brick buffer in MB

Resolution	512×512		1280×800		1600×1024	
Scene	VCT Diff	VCT Spec	VCT Diff	VCT Spec	VCT Diff	VCT Spec
Sponza	17.19	14.07	63.71	52.49	101.19	82.39
Conference	16.24	7.72	60.34	27.55	94.61	43.05
Sibenik	16.17	14.12	60.31	51.38	95.04	81.57
Cornell box	10.86	7.82	25.85	17.76	38.97	30.15
Dungeon	13.91	10.43	54.57	42.28	85.86	67.47

Table 6.7: Performance of the VCT varies with the resolution of final image; The resolution affects only the final step of the algorithm - the cone tracing. All times are in milliseconds.

quirements of brick buffer are shown in table 6.6. With the increasing depth of the octree the memory requirements grow enormously. The representation with gaussian lobes stores less values per node and therefore the brick buffer takes about 30% less memory than the representation with axial radiant intensity.

Shadow map res.	512×512		1024×1024		2048×2048	
Scene	SM render	trav. SM	SM render	trav. SM	SM render	trav. SM
Sponza	1.15	2.69	2.23	10.17	6.88	40.33
Conference	1.27	6.84	2.80	26.66	9.22	105.73
Sibenik	0.86	6.76	2.74	26.62	9.64	105.53
Cornell box	0.41	5.23	1.37	20.14	5.18	79.50
Dungeon	0.57	6.83	2.01	26.63	7.61	105.70

Table 6.8: Performance of the VCT with changes with shadow map resolution; The shadow map resolution affects only the rendering of a shadow map and the traversation of a shadow map; All times are in milliseconds.

Oct. depth	Scene	Oct. build	Brick fill	Dyn. oct. build	Dyn. brick fill	L. inject.	VCT Diff	VCT Spec	Total time
6	Sponza	9.78	5.80	2.74	6.88	10.40	10.86	8.13	39.01
	Conference	8.44	3.68	2.68	6.93	13.98	10.59	7.79	41.97
	Sibenik	12.15	4.26	2.77	7.50	14.44	10.91	9.24	44.86
	Cornell box	8.13	4.56	2.86	8.95	14.07	6.55	4.60	37.04
	Dungeon	7.50	3.97	2.69	7.23	13.24	9.25	8.78	41.19
7	Sponza	16.11	20.59	3.65	13.95	17.80	14.00	12.30	61.69
	Conference	12.14	11.50	3.67	15.59	23.74	13.48	8.19	64.67
	Sibenik	12.94	14.30	3.46	17.11	25.29	13.38	12.86	72.11
	Cornell box	10.35	14.30	3.55	21.34	27.96	6.52	8.49	67.87
	Dungeon	11.42	12.83	3.50	15.56	22.31	11.90	10.83	64.10
8	Sponza	31.81	81.79	4.09	36.86	40.90	17.22	14.09	113.15
	Conference	21.51	48.72	3.96	46.18	55.95	16.05	7.78	129.92
	Sibenik	24.10	60.29	4.60	50.45	59.25	15.92	14.20	144.42
	Cornell box	18.76	51.49	4.92	70.61	77.75	9.87	7.43	170.59
	Dungeon	18.68	45.05	4.75	43.85	48.70	14.73	10.65	122.68

Table 6.9: Performance of the VCT varies with the depth of the voxel octree. All times are in milliseconds.

	VCT - Diff & AO [ms]								
Scene	4	5	6	7	8	9	10	11	12
Sponza	11.67	14.58	17.22	19.35	21.87	23.95	25.93	27.81	29.61
Conference	11.54	14.28	16.05	17.94	20.16	21.09	22.76	23.86	25.20
Sibenik	11.39	13.77	15.92	18.39	20.64	22.27	24.17	25.97	27.53
Cornell box	8.68	9.58	9.87	11.21	11.97	13.23	14.43	15.58	15.12
Dungeon	10.76	12.07	14.73	15.92	17.40	18.67	20.00	21.89	22.73

Table 6.10: Performace of diffuse indirect illumination for various count of diffuse cones; All times are in milliseconds.

Oct. depth	Scene	VFL gen.	Oct. build	Reset bricks	Gather VF	CP inside	Gather ch.	CP edge	Total time	Node Count
6	Sponza	1.77	8.01	0.49	0.16	0.53	3.50	1.12	15.58	6698
	Conference	1.32	7.11	0.22	0.06	0.24	1.93	1.23	12.11	2997
	Sibenik	0.99	11.16	0.30	0.08	0.37	2.33	1.17	16.41	4196
	Cornell box	0.24	7.89	0.35	0.09	0.39	2.57	1.16	12.68	4893
	Dungeon	0.30	7.21	0.29	0.06	0.29	2.28	1.04	11.48	3827
7	Sponza	4.38	11.73	2.03	0.72	2.22	13.29	2.34	36.70	28343
	Conference	2.22	9.92	0.98	0.33	1.11	7.17	1.92	23.64	13116
	Sibenik	2.50	10.44	1.40	0.48	1.57	8.94	1.90	27.23	19539
	Cornell box	0.83	9.52	1.70	0.44	1.46	8.82	1.87	24.65	19045
	Dungeon	0.96	10.46	1.13	0.44	1.18	8.11	1.96	24.25	15231
8	Sponza	12.24	19.56	9.04	3.39	10.16	53.00	6.20	113.60	126639
	Conference	5.80	15.72	4.88	1.59	5.22	32.98	4.05	70.23	58160
	Sibenik	7.40	16.70	6.27	2.29	7.06	39.89	4.78	84.39	87395
	Cornell box	3.11	15.64	5.55	2.04	6.05	33.56	4.29	70.25	77051
	Dungeon	3.38	15.30	4.58	1.62	4.91	30.03	3.91	63.73	62524

Table 6.11: Duration of the steps performed during the voxelization of the static part of the scene; All times are in milliseconds.

Oct. depth	Scene	VFL gen.	Oct. build	Reset bricks	Gather VF	CP inside	Gather ch.	CP edge	Upd. list	Total time
6	Sponza	0.28	2.47	0.43	0.03	0.32	1.72	0.84	3.54	9.62
	Conference	0.36	2.32	0.20	0.04	0.46	1.96	0.83	3.44	9.61
	Sibenik	0.27	2.50	0.27	0.03	0.49	2.29	0.85	3.57	10.27
	Cornell box	0.29	2.57	0.34	0.04	0.83	2.98	0.96	3.79	11.81
	Dungeon	0.29	2.40	0.25	0.03	0.50	2.08	0.87	3.49	9.92
7	Sponza	0.32	3.33	1.82	0.04	1.12	5.20	1.35	4.42	17.59
	Conference	0.49	3.18	0.86	0.07	1.78	6.82	1.62	4.45	19.27
	Sibenik	0.32	3.14	1.26	0.07	1.70	7.91	1.73	4.44	20.57
	Cornell box	0.45	3.10	1.29	0.04	3.12	10.55	1.98	4.37	24.90
	Dungeon	0.38	3.12	1.03	0.05	1.83	6.78	1.58	4.29	19.06
8	Sponza	0.29	3.80	8.05	0.07	3.57	17.15	2.54	5.49	40.95
	Conference	0.55	3.41	3.77	0.04	6.93	26.48	3.60	5.37	50.14
	Sibenik	0.27	4.33	5.61	0.04	6.37	28.02	3.93	6.48	55.05
	Cornell box	0.55	4.37	5.19	0.06	12.82	41.30	5.26	5.99	75.54
	Dungeon	0.45	4.30	4.12	0.13	6.76	23.49	3.41	5.96	48.60

Table 6.12: Duration of the steps performed during the voxelization of the dynamic part of the scene; All times are in milliseconds.

Oct. depth	Scene	SM render	Trav. SM	Insert val.	Upd. list	Reset bricks	CP inside	Gather ch.	CP edge	Total time
6	Sponza	1.15	2.06	0.76	3.19	0.43	0.31	1.68	0.81	10.40
	Conference	1.33	5.03	1.04	3.15	0.19	0.47	1.94	0.82	13.98
	Sibenik	0.93	5.03	1.11	3.36	0.27	0.53	2.38	0.84	14.44
	Cornell box	0.46	3.84	1.19	3.50	0.34	0.83	2.99	0.93	14.07
	Dungeon	0.59	5.04	0.97	3.07	0.25	0.45	2.04	0.83	13.24
7	Sponza	1.15	2.41	0.84	3.99	1.82	1.07	4.91	1.61	17.80
	Conference	1.31	5.88	1.52	4.20	0.86	1.77	6.65	1.55	23.74
	Sibenik	0.90	5.87	1.57	4.05	1.27	1.77	8.17	1.70	25.29
	Cornell box	0.38	4.52	1.61	4.03	1.30	3.25	10.91	1.97	27.96
	Dungeon	0.57	5.93	1.44	3.96	1.02	1.70	6.22	1.46	22.31
8	Sponza	1.18	2.70	1.03	4.85	8.03	3.53	17.07	2.52	40.90
	Conference	1.27	6.79	2.07	4.89	3.80	6.97	26.51	3.66	55.95
	Sibenik	0.90	6.78	2.27	5.49	5.64	6.35	27.99	3.82	59.25
	Cornell box	0.48	5.21	2.23	5.47	5.17	12.76	41.08	5.34	77.75
	Dungeon	0.65	6.81	2.28	5.39	4.19	5.72	20.49	3.16	48.70

Table 6.13: Duration of the steps involved in light parameters injection; All times are in milliseconds.

6.3 Quality analysis

We have compared our solution with the implantation of Imperfect Shadow Maps. The images rendered both with VCT and ISM, and the visualisation of their difference are shown in figure 6.14.

Our VCT implementation produces much darker indirect illumination than the ISM. This issue is probably caused by the transparency of higher level voxels because the voxel intensity is multiplied with the opacity of the voxel. This issue is reduced with the use of directional opacity but is still present. In the Sponza scene, the VCT produces color bleeding effect on the arches above the curtains. This bleeding is not present in the image rendered with ISM. This is caused by several factors. The voxel light behaviour is coarsely approximated. It is approximated either with 6 directional values or with gaussian lobe representation. Another approximation is caused by cone tracing. The integration of the incoming light over hemisphere is approximated by several wide cones. All these approximations produce additional and unrealistic light bleeding.

To compare our solution with Light Propagation Volumes, we have used the NVIDIA's implementation from Direct3D SDK. The images rendered with both LPV and VCT and with similar light and camera properties are shown in the table 6.15.

The LPV, in comparison with the VCT, produces much brighter images. One of the cause is a light leaking. The light leaking is in the LPV worse then in the VCT, since the LPV uses coarse grid. The LPV uses hierarchical grids, which are repositioned according to camera. When the camera moves, there are visible artefacts on the edge of the image.

The ambient occlusion better shows the problems of the cone tracing. The images rendered with ambient occlusion and various depths of the octree are shown in table 6.20. With increasing depth of the octrees, the VCT is able to render more details. On the other hand, the discontinuities mentioned in

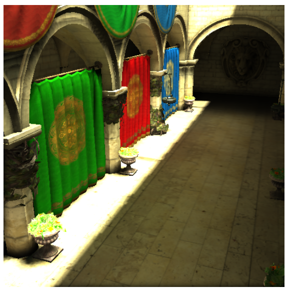
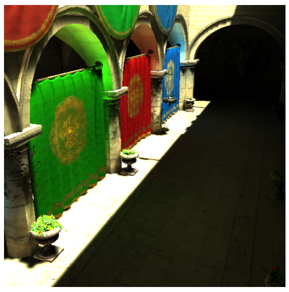
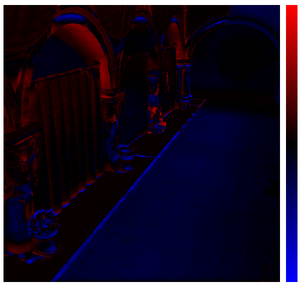
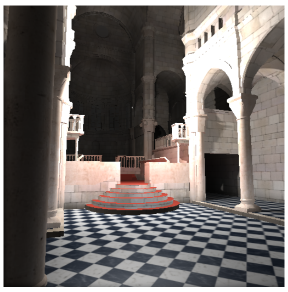
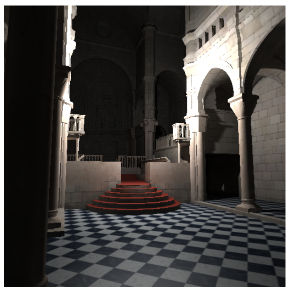
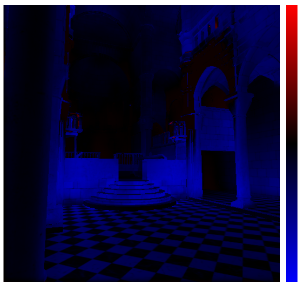
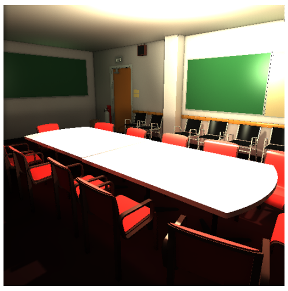
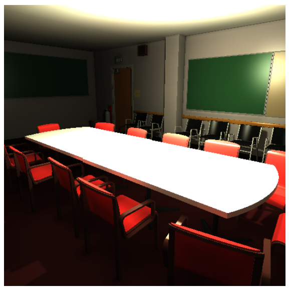
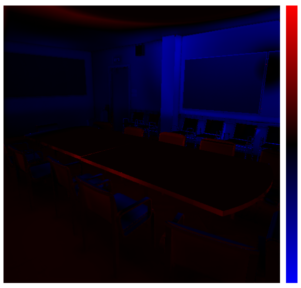
ISM - 1024 VPL	VCT	difference
		 0.318 -0.298
		 0.244 -0.221
		 0.321 -0.305

Table 6.14: The difference of the brightness between Imperfect Shadow Maps algorithm with 1024 lights and our VCT implementation

section 5.4.2 are more frequent with the increasing depth of the octree. These discontinuities produce visible artefacts.

The Cornell box scene shows another issue of the VCT. The scene is approximated by the voxel octree. A voxel of the octree is cube with the sides aligned to 3 main axes. When the scene contain flat surface not aligned with these axes, the voxels of the octree form steps to represent this surface. When the indirect illumination is computed for the points on this surface, the samples for one surface point may fall in the fully opaque voxel, while the samples for adjacent surface point may fall into empty space. This shows in the rendered image as a dark stripe.

Table 6.17 shows the comparison of the two implemented voxel representation. Both solution produce plausible indirect illumination. The representation using gaussian lobes produces minor artefacts, which are shown in figure 6.1. From our observation this issue is caused by the inconsistency of normals stored in bricks. The voxel-fragments generated on the thin wall can have normals facing opposite directions. When these normals are stored and filtered in the bricks, then the normal value creates splodges on

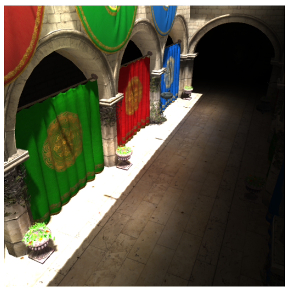
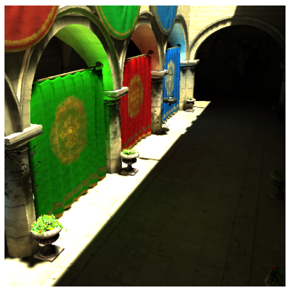
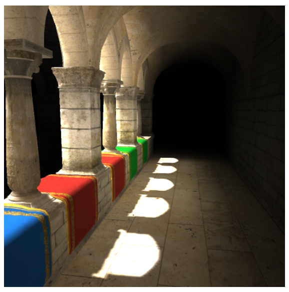
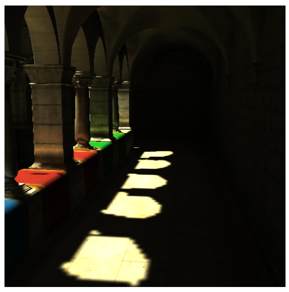
LPV	VCT
	
	

Table 6.15: The Sponza scene rendered using LPV and VCT

such a thin wall as shown in figure 6.1.

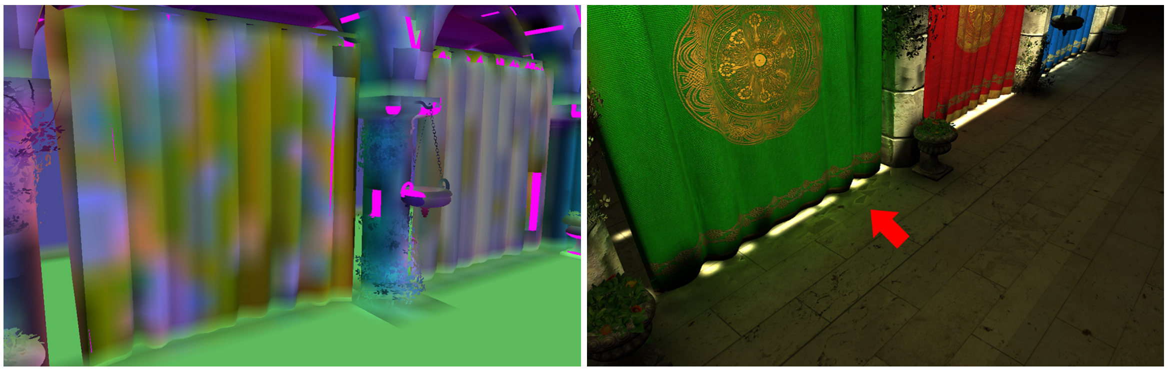


Figure 6.1: The normals stored in the bricks can form splodges on thin walls (on the left) and thus created indirect illumination artefacts (on the right)

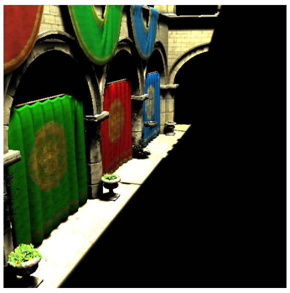
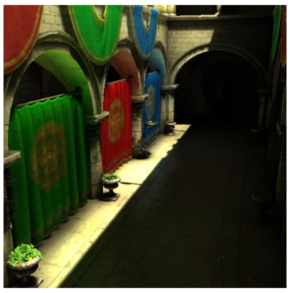
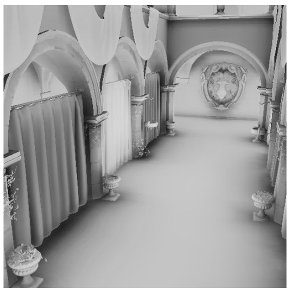
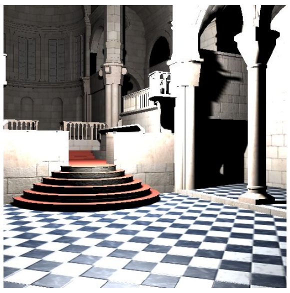
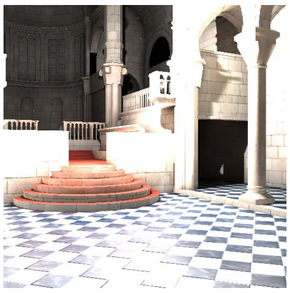
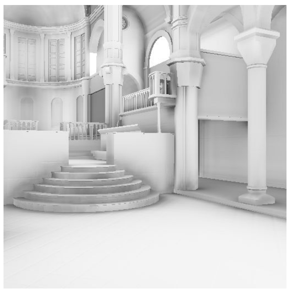
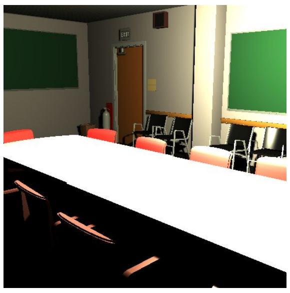
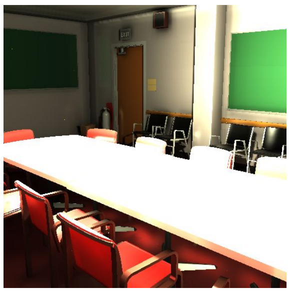
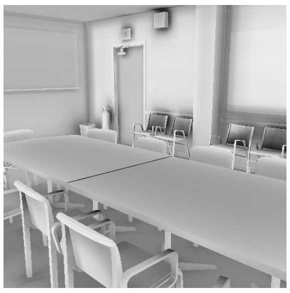
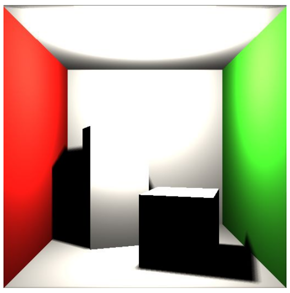
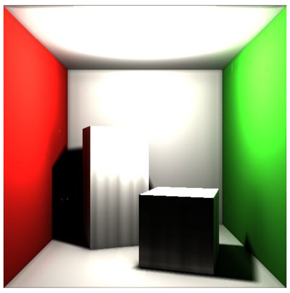
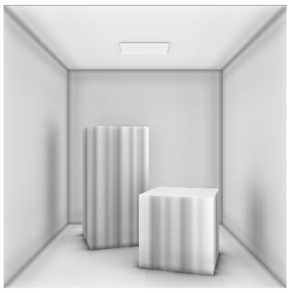
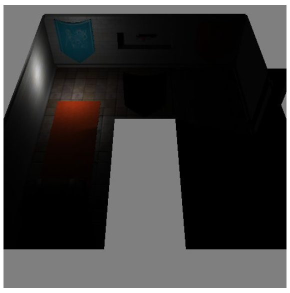
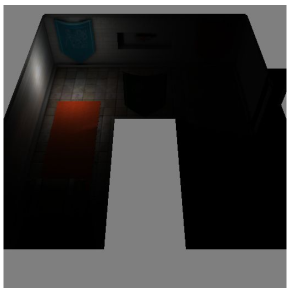
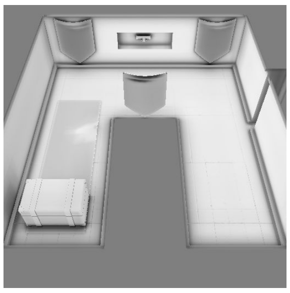
Direct illumination only	Direct + Indirect + AO	AO
		
		
		
		
		

Table 6.16: The tested scenes rendered with direct illumination (on the left), global illumination with ambient occlusion (in the middle) and with ambient occlusion (on the right)

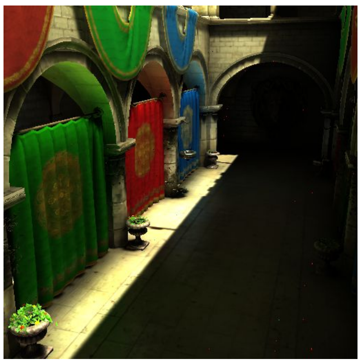
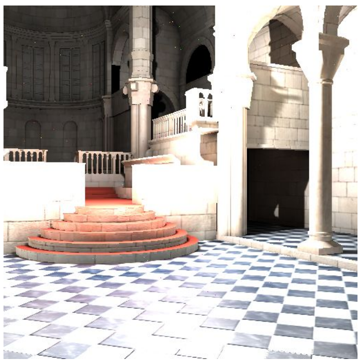
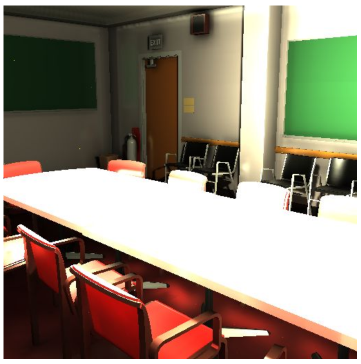
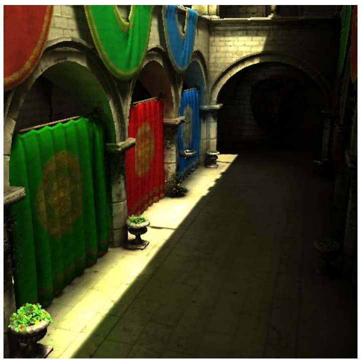
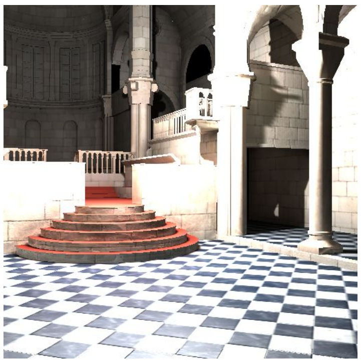
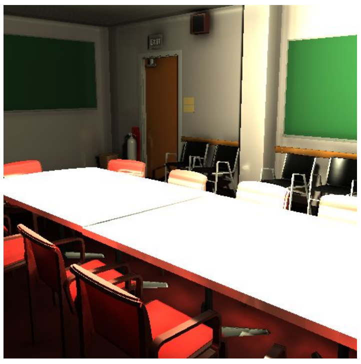
6 axial radiant intensity		
		
Gauss. lobes		
		

Table 6.17: The comparison of the images rendered with VCT using two type of voxel representation

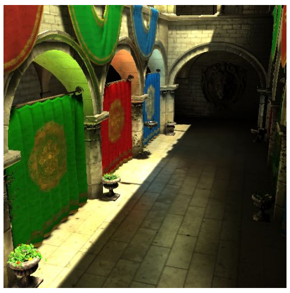
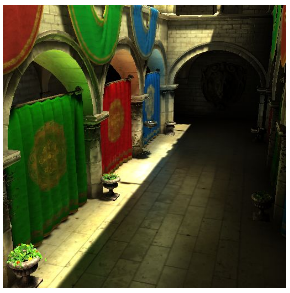
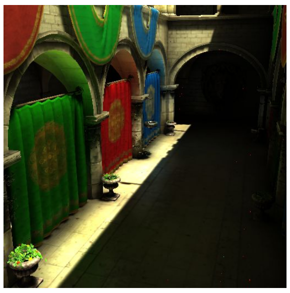
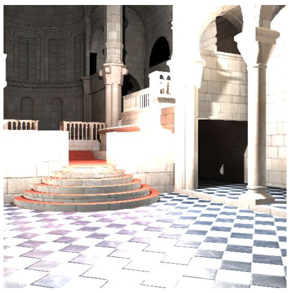
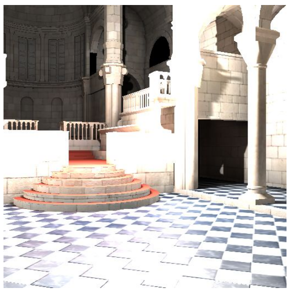
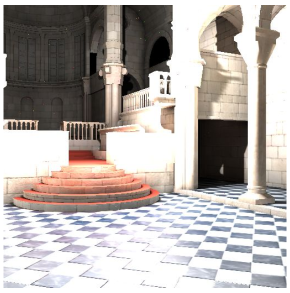
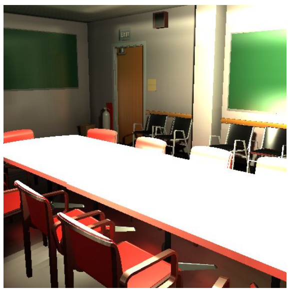
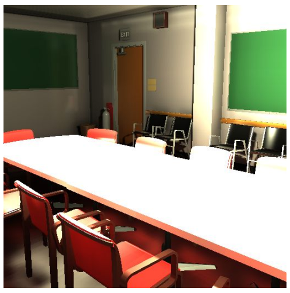
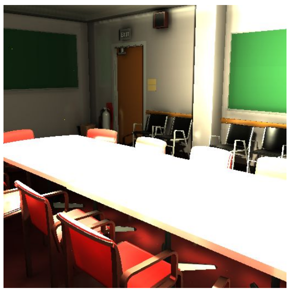
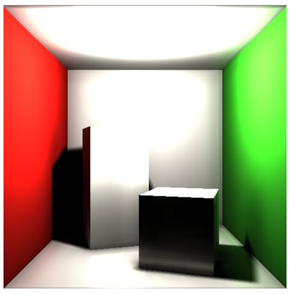
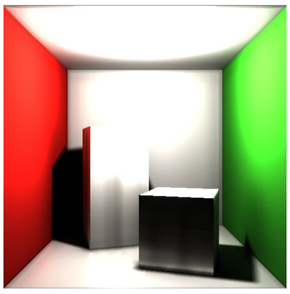
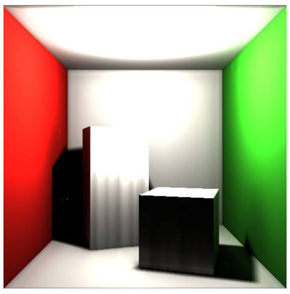
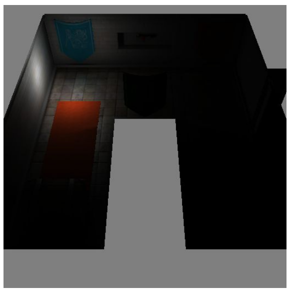
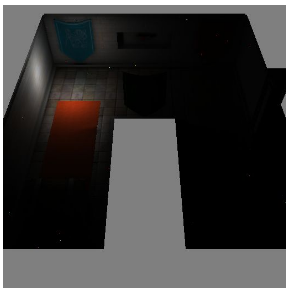
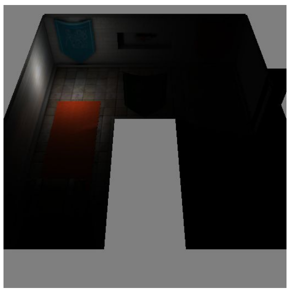
Octree Depth		
6	7	8
		
		
		
		
		

Table 6.18: The tested scenes rendered with global illumination and ambient occlusion with the various depth of the octree

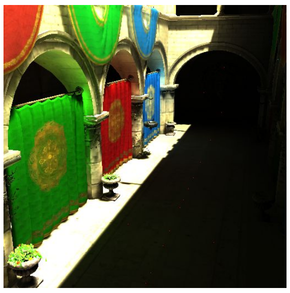
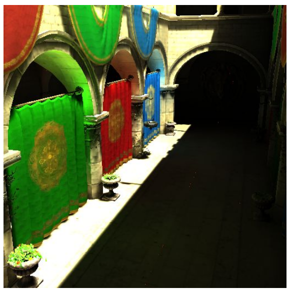
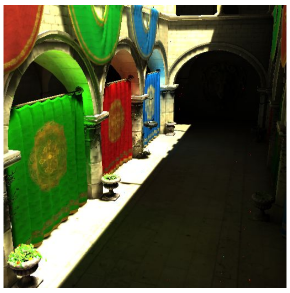
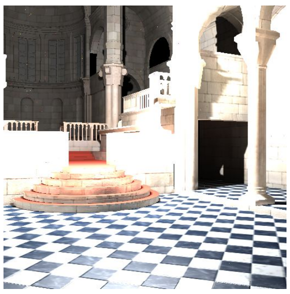
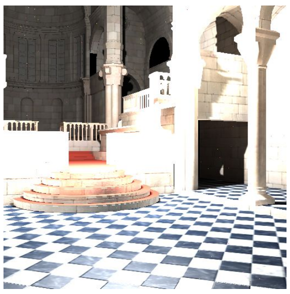
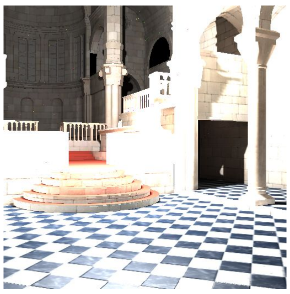
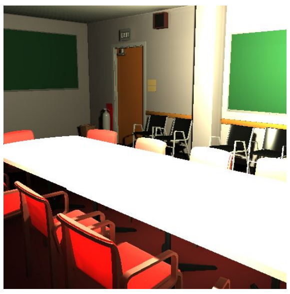
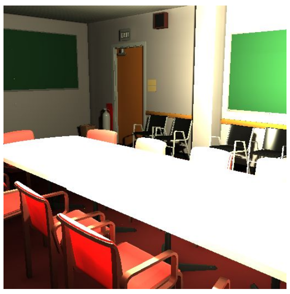
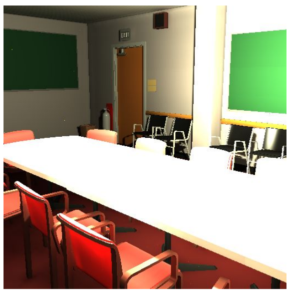
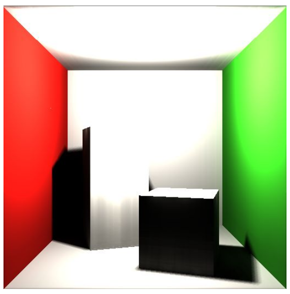
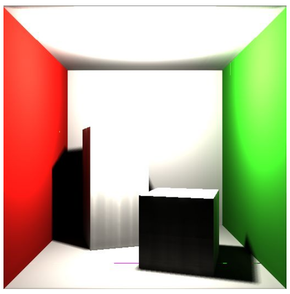
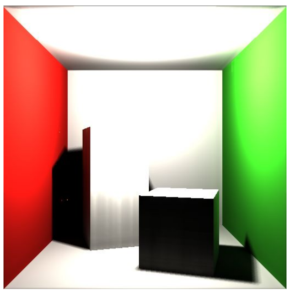
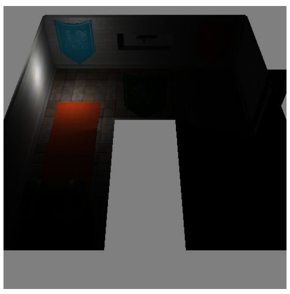
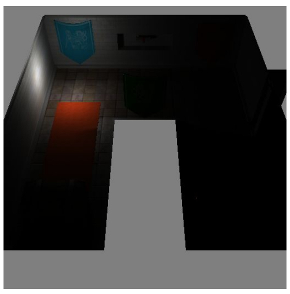
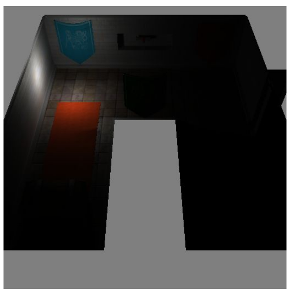
Cone count		
4	5	6
		
		
		
		
		

Table 6.19: The tested scenes rendered with global illumination with the various number of diffuse cones

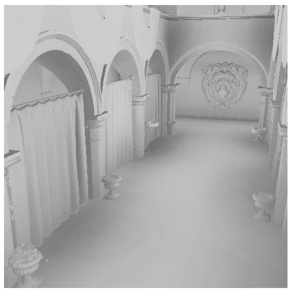
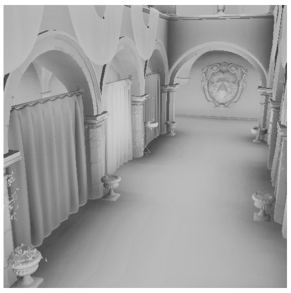
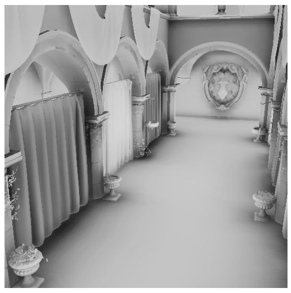
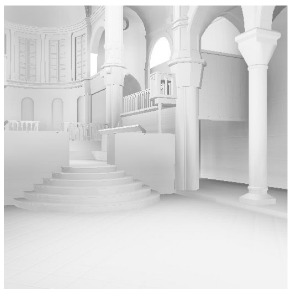
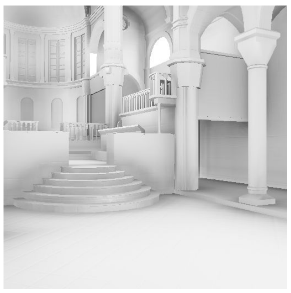
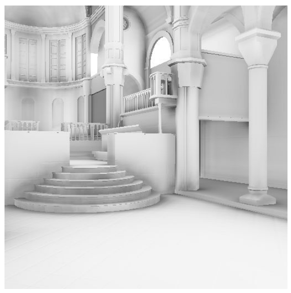
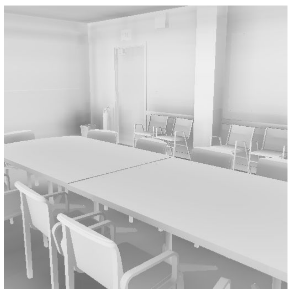
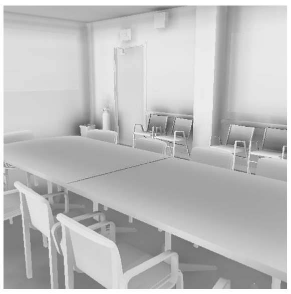
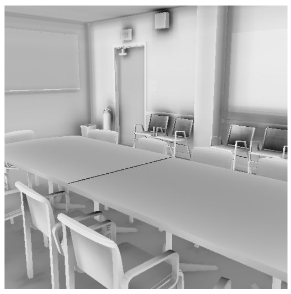
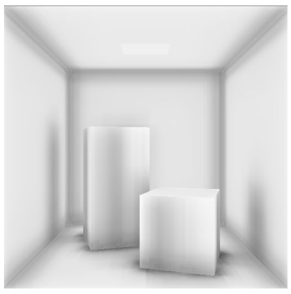
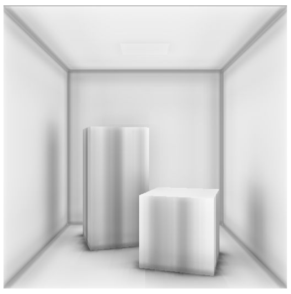
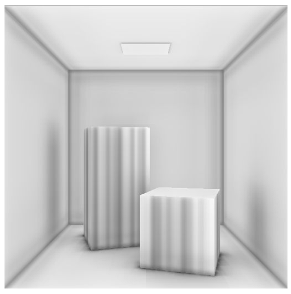
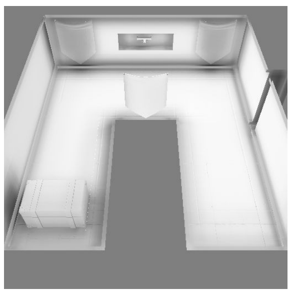
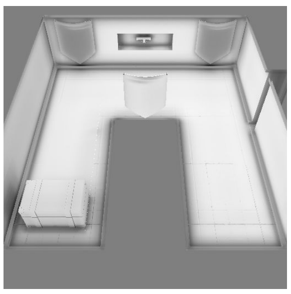
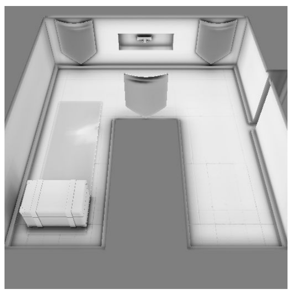
Octree Depth		
6	7	8
		
		
		
		
		

Table 6.20: The tested scenes rendered with ambient occlusion and with the various depth of the octree

7. Conclusion

In this thesis we have reviewed the indirect illumination algorithms used for real-time applications. We have implemented the Voxel Cone Tracing algorithm [Cra+11]. This algorithm uses sparse voxel octree as the simplified representation of the scene. For each point of the surface, the diffuse and specular indirect illumination is gathered from this octree. This gathering is performed by tracing several cones, that cover the area of hemisphere around the surface point. This tracing is realized by sampling. For each sample we perform a lookup into the octree at the depth equal to the diameter of the sample.

7.1 Summary

Our implementation successfully provides coarse approximation of indirect illumination. The implementation is capable of rendering diffuse and specular indirect illumination, ambient occlusion and is also capable of rendering dynamic objects. The overall quality is comparable with both ISM and LPV algorithms. Above that, the specular reflections of our implementation significantly improve the image quality. Although the algorithm performs the steps, which serve to maintain the continuity of values between two adjacent voxels, some discontinuities still remains. These discontinuities cause artefacts, which affect the quality of final image. The implementation of voxel cone tracing part of the algorithm provides from interactive to real-time rendering speed for small resolutions.

Another source of the artefacts is sampling. For tight cones that are used for specular reflections it is necessary to place successive samples very close next to each other to get plausible results. This means it is necessary to use more samples and thus reduce the rendering frame-rate.

The update of the voxel octree with the light related values involves many expansive steps with many memory accesses. This filtering proves to be major bottleneck of entire method. We have implemented two different representations of voxel light information. The representation using the gaussian lobes uses less parameters. This means it uses less memory and light filtering step is faster. On the other hand the representation with 6 values of axial light intensity produces more stable indirect illumination with less errors.

7.2 Future work

Our implementation shows promising results, but it would benefit from several improvements. One of the biggest issue, as seen in the section 6.3, is the light leaking through the walls. This is caused by the transparency of the higher level voxels. In current implementation the scene is expressed by boundary representation and this representation is also projected to the created voxels. In future work it would be beneficial to device new voxelization method, which would create also the voxels inside of the objects.

Another issue of our voxel cone tracing is the sampling. Sampling is used to acquire the necessary values from the voxel octree along the direction of the cone. The distance between two successive samples is often too big and the part of the voxel with significant opacity is missed. To address this issue, we divide this distance and make necessary adjustments, which leads to multiplying the number of samples. In future work, our implementation would benefit from implementing the voxel cone tracing as true sparse octree tracing [LK10].

The diffuse indirect illumination is low frequency information. The performance of the VCT could be significantly improved if the diffuse indirect illumination was computed on a resolution lower than the resolution of the final image. Such a optimization would require edge detection pass, since the indirect illumination on the edges must be computed in full resolution. This optimization effectiveness is reduced when normal maps are used, since normal maps cause the frequent normal discontinuity of two adjacent surface points.

As the process of filtering the octree is main bottleneck of our implementation, this process would benefit from substantial optimization. The obvious candidate for this optimization is the step where brick values are averaged from its child bricks. This step takes around 50% of the whole light injection process.

Bibliography

[ASS13] Assimp developers. *Assimp C++ library*. 2013. URL: `http://assimp.sourceforge.net/`.

[Boo13] Boost developers. *Boost C++ library*. 2013. URL: `http://www.boost.org/`.

[Bro03] PA Bromiley. "Products and convolutions of Gaussian distributions". In: *Medical School, Univ. Manchester, Manchester, UK, Tech. Rep* 3 (2003), p. 2003.

[CG12] Cyril Crassin and Simon Green. "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer". In: *OpenGL Insights*. Ed. by Patrick Cozzi and Christophe Riccio. CRC Press, July 2012, pp. 303–319. ISBN: 978-1439893760. URL: `http://www.openglinsights.com/`.

[CLT07] K. Crane, I. Llamas, and S. Tariq. "Real-time simulation and rendering of 3d fluids". In: *GPU Gems*. Vol. 3. Addison-Wesley Professional, 2007, pp. 633–675. ISBN: 978-0321515261.

[COM98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. "Appearance-preserving simplification". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM. 1998, pp. 115–122.

[Cra+11] Cyril Crassin et al. "Interactive indirect illumination using voxel cone tracing". In: *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, pp. 1921–1930.

[CUDA13] NVIDIA Corporation. *CUDA parallel computing platform*. 2013. URL: `http://www.nvidia.com/object/cuda_home_new.html`.

[Dee+88] Michael Deering et al. "The triangle processor and normal vector shader: a VLSI system for high performance graphics". In: *ACM SIGGRAPH Computer Graphics*. Vol. 22. 4. ACM, 1988, pp. 21–30.

[DS05] Carsten Dachsbacher and Marc Stamminger. "Reflective shadow maps". In: *Proceedings of the 2005 ACM SIGGRAPH symposium on Interactive 3D graphics and games*. ACM. 2005, pp. 203–231.

[DX13] Microsoft. *Microsoft DirectX*. 2013. URL: `http://msdn.microsoft.com/library/windows/apps/hh452744.aspx`.

[Eve01] Cass Everitt. "Interactive order-independent transparency". In: *White paper, nVIDIA* 2.6 (2001), p. 7.

[Fou92] Alain Fournier. "Normal Distribution Functions and Multiple Surfaces". In: *GI '92 Workshop on Local Illumination* (1992).

[GLEW13] GLEW developers. *GLEW C/C++ library*. 2013. URL: `http://glew.sourceforge.net/`.

[GLFW13] GLFW developers. *GLFW C library*. 2013. URL: `http://www.glfw.org/`.

[GLM13] G-Truc Creation. *GLM C++ library*. 2013. URL: `http://glm.g-truc.net/`.

[GLSL] The Khronos Group. *OpenGL Shading Language (GLSL) Reference Pages*. 2013. URL: `http://www.opengl.org/sdk/docs/manglsl/`.

[GLUT13] GLUT developers. *GLUT library*. 2013. URL: `http://www.opengl.org/resources/libraries/glut/`.

[ICG86] David S Immel, Michael F Cohen, and Donald P Greenberg. “A radiosity method for non-diffuse environments”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 20. 4. ACM. 1986, pp. 133–142.

[Jen96] Henrik Wann Jensen. “Global illumination using photon maps”. In: *Rendering Techniques* 96 (1996), pp. 21–30.

[Kaj86] James T Kajiya. “The rendering equation”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 20. 4. ACM, 1986, pp. 143–150.

[Kap09] Anton Kaplanyan. “Light propagation volumes in cryengine 3”. In: *Advances in Real-Time Rendering in 3D Graphics and Games Course–SIGGRAPH* (2009).

[KD10] Anton Kaplanyan and Carsten Dachsbacher. “Cascaded light propagation volumes for real-time indirect illumination”. In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM. 2010, pp. 99–107.

[Kel97] Alexander Keller. “Instant radiosity”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1997, pp. 49–56.

[KIX12] Kixor developers. *Kixor library*. 2012. URL: `http://www.kixor.net/dev/objloader/`.

[KL96] Venkat Krishnamurthy and Marc Levoy. “Fitting smooth surfaces to dense polygon meshes”. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 313–324.

[Laf96] Eric Lafortune. “Mathematical models and Monte Carlo algorithms for physically based rendering”. PhD thesis. Citeseer, 1996.

[LHN05] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. “Octree Textures on the GPU”. In: *GPU Gems 2*. Addison-Wesley Professional, 2005. ISBN: 0-321-33559-7.

[LK10] Samuli Laine and Tero Karras. “Efficient sparse voxel octrees–analysis, extensions, and implementation”. In: *NVIDIA Corporation* 2 (2010).

[LOBJ13] LibObj developers. *LibObj C++ library*. 2013. URL: `http://sourceforge.net/projects/libobj/`.

[Mea80] Donald JR Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[MKC07] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. “Efficient point-based rendering using image reconstruction”. In: *PBG’07: Proceedings of the Eurographics Symposium on Point-Based Graphics*. 2007, pp. 101–108.

[ML09] Morgan McGuire and David Luebke. “Hardware-accelerated global illumination by image space photon mapping”. In: *Proceedings of the Conference on High Performance Graphics 2009*. ACM. 2009, pp. 77–89.

[Nic65] Fred E Nicodemus. “Directional reflectance and emissivity of an opaque surface”. In: *Applied Optics* 4.7 (1965), pp. 767–773.

[OGL4] The Khronos Group. *OpenGL 4 Reference Pages*. 2013. URL: `http://www.opengl.org/sdk/docs/man/xhtml/`.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123750792, 9780123750792.

[Pho75] Bui Tuong Phong. “Illumination for computer generated pictures”. In: *Communications of the ACM* 18.6 (1975), pp. 311–317.

[Rit+08] Tobias Ritschel et al. “Imperfect shadow maps for efficient computation of indirect illumination”. In: *ACM Transactions on Graphics (TOG)*. Vol. 27. 5. ACM. 2008, p. 129.

[Rit+11] Tobias Ritschel et al. “Making Imperfect Shadow Maps View-Adaptive: High-Quality Global Illumination in Large Dynamic Scenes”. In: *Computer Graphics Forum*. Wiley Online Library. 2011.

[Rit+12] T. Ritschel et al. “The state of the art in interactive global illumination”. In: *Computer Graphics Forum*. Vol. 31. 1. Wiley Online Library. 2012, pp. 160–188.

[RSC87] William T Reeves, David H Salesin, and Robert L Cook. “Rendering antialiased shadows with depth maps”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 21. 4. ACM. 1987, pp. 283–291.

[ST90] Takafumi Saito and Tokiichiro Takahashi. “Comprehensible rendering of 3-D shapes”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 24. 4. ACM. 1990, pp. 197–206.

[Sto+04] William A Stokes et al. “Perceptual illumination components: a new approach to efficient, high quality global illumination rendering”. In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. ACM, 2004, pp. 742–749.

[Tok05] Michael Toksvig. “Mipmapping normal maps”. In: *Journal of Graphics, GPU, and Game Tools* 10.3 (2005), pp. 65–71.

[Wil78] Lance Williams. “Casting curved shadows on curved surfaces”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 12. 3. ACM. 1978, pp. 270–274.

Appendix A: List of Abbreviations

AO	Ambient Occlusion
BRDF	Bidirectional Reflectance Distribution Function
BRSM	Bidirectional Reflective Shadow Maps
BSDF	Bidirectional Scattering Distribution Function
BTDF	Bidirectional Transmittance Distribution Function
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFS	Depth-First Search
GUI	Graphical User Interface
GPU	Graphics Processing Unit
ISM	Imperfect Shadow Maps
LPV	Light Propagation Volumes
NDF	Normal Distribution Function
GLSL	OpenGL Shading Language
GV	Geometry Volume
PCF	Percetage-Closer Filtering
pVPL	potential Virtual Point Light
RSM	Reflective Shadow Maps
SDK	Software Development Kit
SM	Shadow Map
VPL	Virtual Point Light
VCT	Voxel Cone Tracing

Appendix B: Additional images

The contrast and brightness of the following images have been modified.

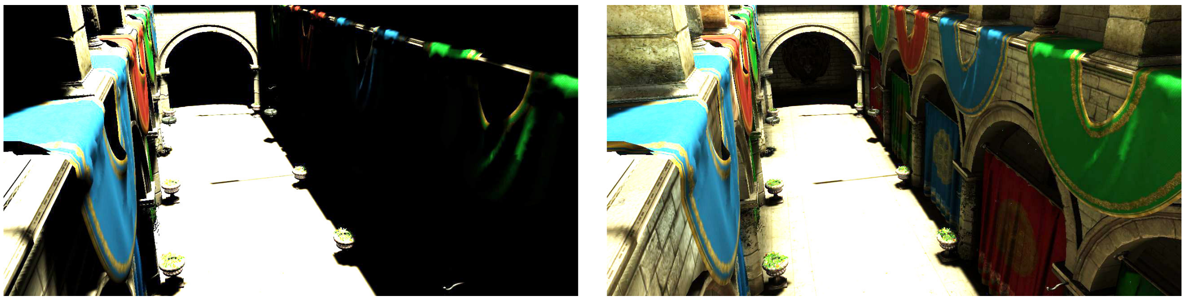


Figure B.1: Sponza scene without indirect illumination(on the left) and with indirect illumination(on the right)

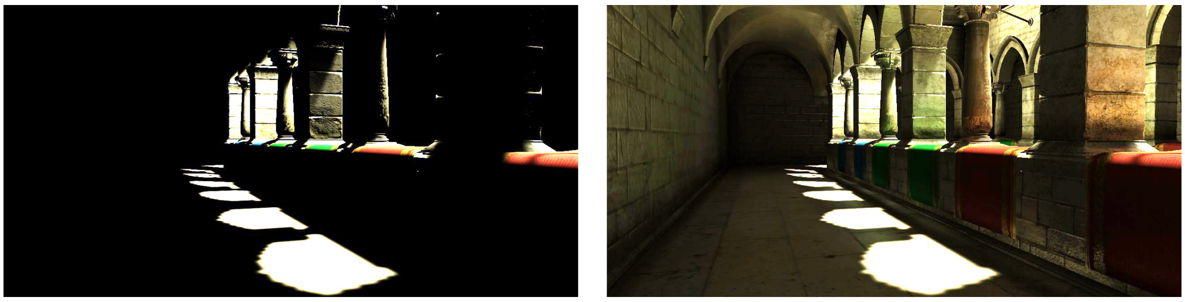


Figure B.2: Sponza scene without indirect illumination(on the left) and with indirect illumination(on the right)

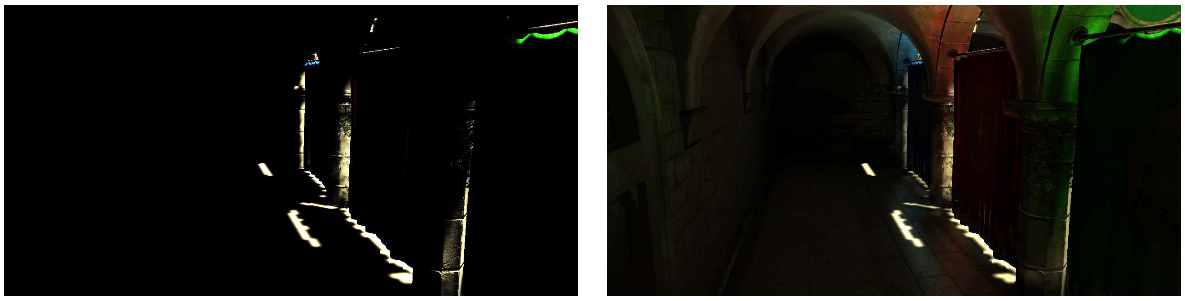


Figure B.3: Sponza scene without indirect illumination(on the left) and with indirect illumination(on the right)

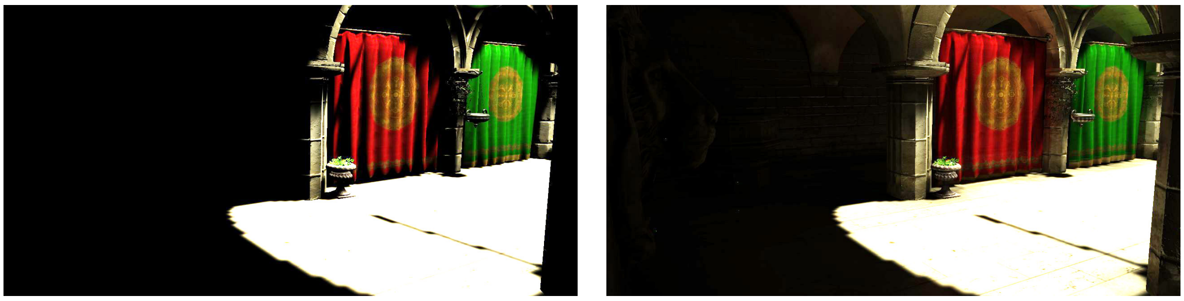


Figure B.4: Sponza scene without indirect illumination(on the left) and with indirect illumination(on the right)

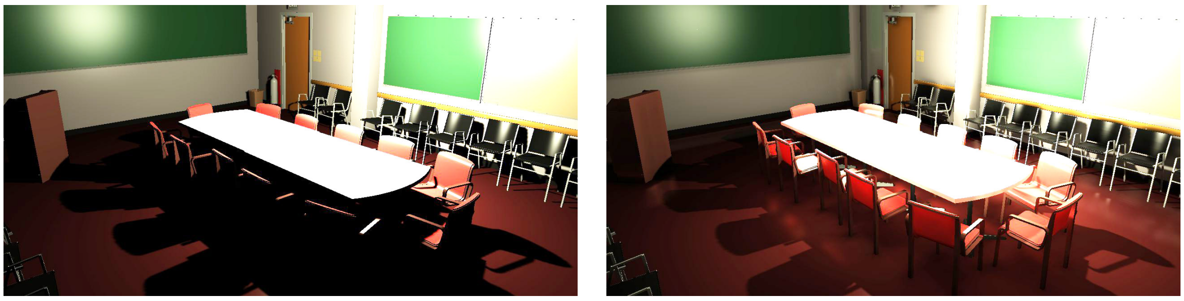


Figure B.5: Conference scene without indirect illumination(on the left) and with indirect illumination(on the right)

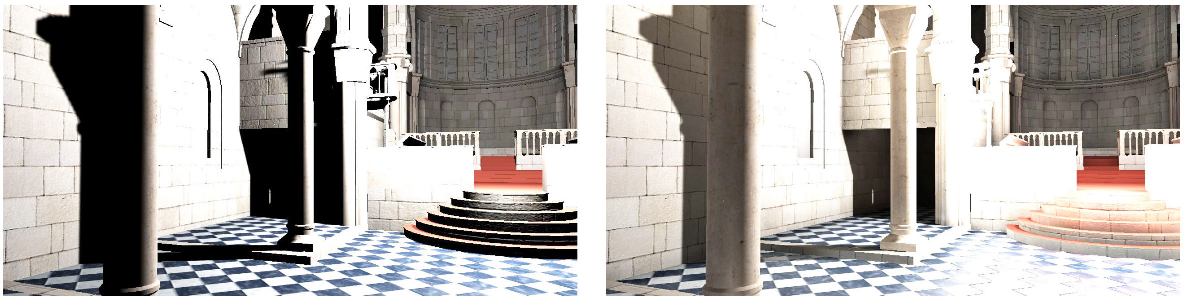


Figure B.6: Sibenik scene without indirect illumination(on the left) and with indirect illumination(on the right)

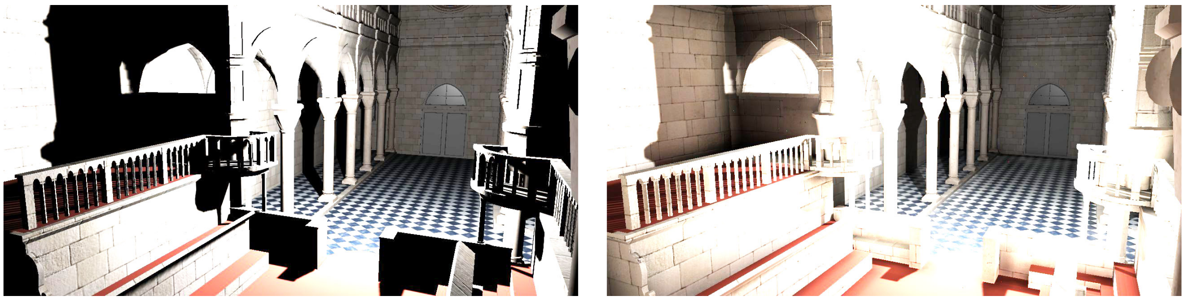


Figure B.7: Sibenik scene without indirect illumination(on the left) and with indirect illumination(on the right)

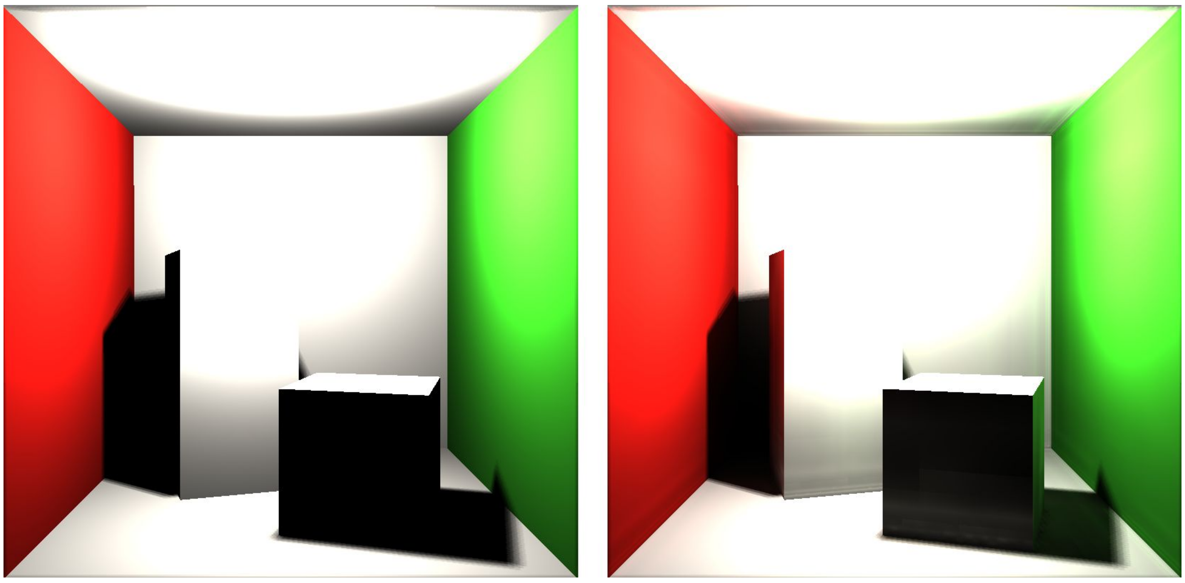


Figure B.8: Cornell box scene without indirect illumination(on the left) and with indirect illumination(on the right)

Appendix C: User manual and installation

The application is shipped with both source code and the binaries for the windows platform. The executables for the other platforms must be compiled.

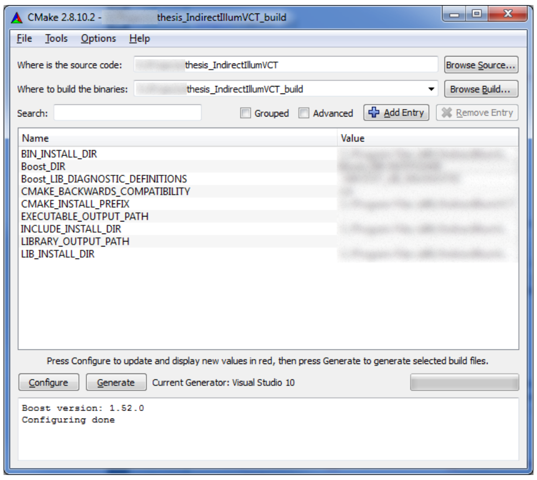
C.1 Compilation

The application uses several libraries, which are stated in table C.1. Most of the libraries are included in the project and are not necessary to be installed. The only library which is not included is the Boost library.

Library	Version	Attached
OpenGL	4.2	yes
ASSIMP	3.0.1270	yes
GLEW	1.9.0	yes
GLFW	2.7.6	yes
GLM	0.9.3.4	yes
Boost	1.5.2	no

Table C.1: List of the libraries needed for the compilation

The project is set using CMake, the cross-platform build system. The CMake guides the user, through the process of project creation, with *cmake-gui*. *Cmake-gui*, shown in the figure C.1, allows the user to choose the compiler and alerts the user about possible dependency issues.



CMake 2.8.10.2 - [redacted]thesis_IndirectIllumVCT_build

Where is the source code: [redacted]thesis_IndirectIllumVCT

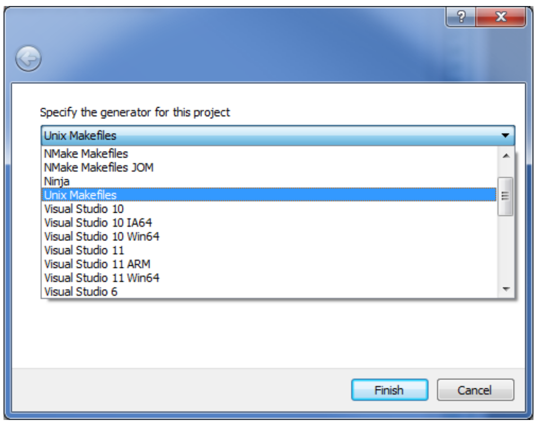
Where to build the binaries: [redacted]thesis_IndirectIllumVCT_build

Name	Value
BIN_INSTALL_DIR	[redacted]
Boost_DIR	[redacted]
Boost_LIB_DIAGNOSTIC_DEFINITIONS	[redacted]
CMAKE_BACKWARDS_COMPATIBILITY	[redacted]
CMAKE_INSTALL_PREFIX	[redacted]
EXECUTABLE_OUTPUT_PATH	[redacted]
INCLUDE_INSTALL_DIR	[redacted]
LIBRARY_OUTPUT_PATH	[redacted]
LIB_INSTALL_DIR	[redacted]

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 10

```
Boost version: 1.52.0
Configuring done
```



Specify the generator for this project

Unix Makefiles

- NMake Makefiles
- NMake Makefiles JOM
- Ninja
- Unix Makefiles
- Visual Studio 10
- Visual Studio 10 IA64
- Visual Studio 10 Win64
- Visual Studio 11
- Visual Studio 11 ARM
- Visual Studio 11 Win64
- Visual Studio 6

Figure C.1: The cmake-gui application (on the left); Cmake lets user to choose a compiler (on the right)

The source directory must be set to the base directory of the source, where the file CMakeLists.txt is located. Set the build directory and press the *Configure* button. Then, the cmake-gui will let user to choose the compiler. Press *Generate* button and the project for the selected compiler will be generated.

For the Unix-platform, enter following commands:

```
cd build_directory/
make
```

For the Windows platform and Microsoft Visual Studio IDE, Cmake generates the solution. Open it it in the Visual Studio. The solution consist of 5 projects. Set the IndirectIllumnVCT as the main project and press the key F7 to build the whole solution.

C.2 Usage

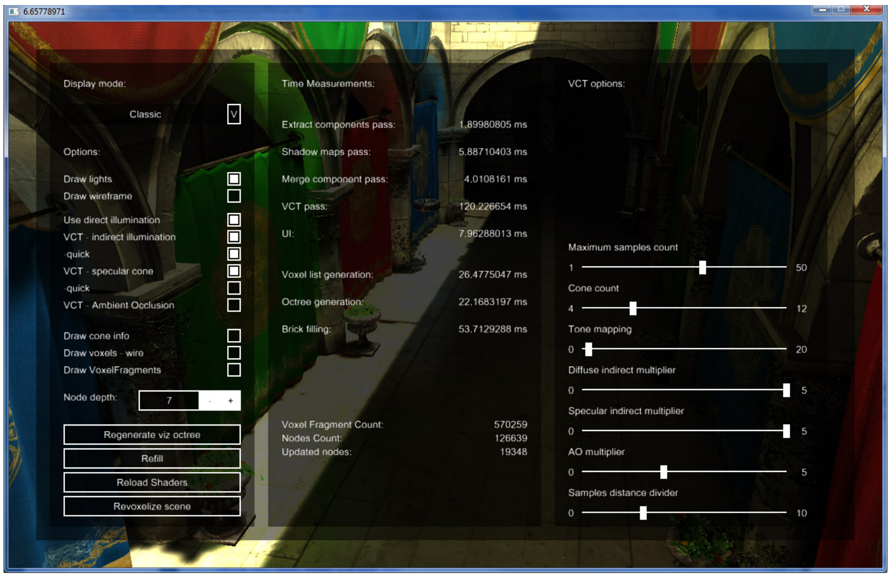
The application executable for the windows platform is located in directory /win . The program has several command line parameters, that are shown in table C.2. If these parameters are not set, the default values will be used. The order of the parameters is arbitrary. All the other parameters of VCT are set with GUI.

Parameter swith	Default value	Accepted values	Description
-d	0.2f	float	Sets the portion of dynamic bricks
-vd	8	integer 1-8	Sets the depth of voxel octree
-s	sponza.conf	string	Sets the path to the scene configuration file
-smRes	512	integer < GL_MAX_TEXTURE_SIZE	Sets the resolution of the shadow maps
-qatt	1	0 or 1	Sets the renderer to use quadratic attenuation
-nm	1	0 or 1	Sets the renderer to use normal maps
-pcf	1	0 or 1	Sets the renderer to use Percentage-closer filtering
-wmm	8	integer	Sets the level of texture mip-map used during voxel-fragment generation
-viewW	1280	integer < GL_MAX_TEXTURE_SIZE	Sets the width of the viewport
-viewH	800	integer < GL_MAX_TEXTURE_SIZE	Sets the height of the viewport
-windowW	1280	integer < GL_MAX_TEXTURE_SIZE	Sets the width of the window. The minimum value for correct GUI is 1280.
-windowH	800	integer < GL_MAX_TEXTURE_SIZE	Sets the height of the window. The minimum value for correct GUI is 800.

Table C.2: The command line parameters of the program.

Once the program is running, and a scene is loaded, the user can rotate the camera using a mouse. The movement of the camera is mapped onto the *arrow* keys. *PgUp* and *PgDown* keys are used to set the height of the camera. The field of view of the camera is set with + and - keys. The movement of the light is mapped onto keys *NumPad 8, NumPad 2, NumPad 4, NumPad 6*. The elevation of the light is set by the keys *NumPad 7* and *NumPad 9*.

The GUI of the program is shown and hidden by the *space* key. The program gets closed after the user hit *escape* key. If the scene contain animations, the dynamic object animations are run and stopped by *M* key and the light animations are run and stopped by *L* key.



```
6.65778971

Display mode:                          Time Measurements:                              VCT options:
Classic  V
                                       Extract components pass:   1.89980805 ms
Options:                               Shadow maps pass:          5.88710403 ms
Draw lights                  ✓         Merge component pass:      4.0108161 ms
Draw wireframe               ☐         VCT pass:                  120.226654 ms
Use direct illumination      ✓         UI:                        7.96288013 ms
VCT - indirect illumination  ✓                                                         Maximum samples count
-quick                       ✓                                                         1  50
VCT - specular cone          ✓         Voxel list generation:     26.4775047 ms        Cone count
-quick                       ☐         Octree generation:         22.1683197 ms        4  12
VCT - Ambient Occlusion      ☐         Brick filling:             53.7129288 ms        Tone mapping
                                                                                       0  20
Draw cone info               ☐                                                         Diffuse indirect multiplier
Draw voxels - wire           ☐                                                         0  5
Draw VoxelFragments          ☐                                                         Specular indirect multiplier
Node depth:  7  -  +                   Voxel Fragment Count:      570259               0  5
                                       Nodes Count:               126639               AO multiplier
Regenerate viz octree                  Updated nodes:             19348                0  5
Refill                                                                                 Samples distance divider
Reload Shaders                                                                         0  10
Revoxelize scene
```

Figure C.2: The GUI of the program

The GUI, shown in figure C.2, is divided into 3 sections.

In the left section, the user can turn on/off direct illumination, indirect illumination and ambient occlusion. There are also options used for debugging purposes, like *draw voxels* and *draw cone info*. The combo box located on the top of this section allows user to switch render style to voxel-fragment = color mode. In this mode, the selected value of the node is directly mapped onto the underlying surface. The middle section of the GUI is dedicated to the measurements. The parameters of the cone tracing and tonemapping are located on the right side of the GUI. A user can choose the number of the cones, the number of the samples for a cone and the distance between two succeeding samples. Additionally, the user can tweak the impact of all the indirect illumination components.

Appendix D: CD content

`thesis_drinotom/`	the root project directory
`external`	external libraries
`IndirectIllumVCT`	the implementation directory
`data`	testing scenes and additional data
`shaders`	shaders for VCT
`ui`	shaders for GUI
`source`	source codes
`text`	source files and PDF of this thesis
`win32release`	application for Windows platform
`README.TXT`	build instruction