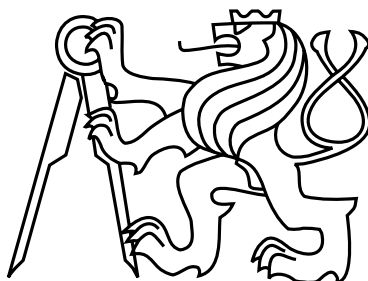


Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Master's Thesis

Global Illumination via Instant Radiosity

Bc. Tomáš Barák

Supervisor: doc. Ing. Vlastimil Havran, Ph.D.

Study Programme: Open Informatics

Field of Study: Computer Graphics and Interaction

May 11, 2012

Acknowledgements

I am very grateful for the cooperation of my supervisor doc. Ing. Vlastimil Havran, Ph.D., his frequent comments helped me a lot with the creation of this thesis. I would also like to express my thanks to Ing. Jiří Bittner, Ph.D. for his numerous remarks and ideas. Above all, I deeply appreciate the help of my girlfriend, since she gave me irreplaceable support.

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 č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 11, 2012

.....

Abstract

Global illumination algorithms form a fundamental part of the realistic image synthesis discipline. They consider the necessary indirect illumination component, and they usually produce highly plausible images. On the other hand, the precise evaluation of the indirect illumination tends to be the most time consuming part of the rendering process. Huge speedup can be achieved with the use of stochastic illumination estimators.

An example of the stochastic approach can be found in the Instant Radiosity algorithms. We explore the family of the Instant Radiosity algorithms focusing on the Imperfect Shadow Maps (ISM) method, which was introduced by Ritschel et al. in 2008. The subject of the submitted thesis is the implementation of the ISM algorithm with a consideration of the possibilities of a modern GPU (2011). Our ISM based renderer produces satisfactory images in real-time frame rates.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Subject of this Thesis	2
1.3	Thesis Structure	2
2	Theory behind the Radiosity Methods	5
2.1	Scene Representation	5
2.2	Important Radiometric Quantities	5
2.3	Bidirectional Reflectance Distribution Function	6
2.4	Global Illumination	6
2.5	Radiosity Methods	7
2.6	Quasi-Monte Carlo Estimation	8
3	Common Rasterization Techniques	9
3.1	Shadow Mapping	9
3.1.1	Point Light Sources	9
3.1.2	Area Light Sources	11
3.2	Deferred Rendering	11
3.3	Geometry Culling	12
4	Family of the Instant Radiosity Algorithms	15
4.1	Instant Radiosity	15
4.2	Reflective Shadow Maps	17
4.3	Splatting Indirect Illumination	17
4.4	Incremental Instant Radiosity	18
4.5	Coherent Shadow Maps	19
4.6	Coherent Surface Shadow Maps	20
4.7	VPL Sampling Improvements	21
4.7.1	Simple Iterative Method	21
4.7.2	Bidirectional Instant Radiosity	21
4.8	Precomputed Radiance Map	21
4.9	Implicit Visibility and Antiradiance	22
4.10	Combination of Global and Local VPLs	22
4.11	Imperfect Shadow Maps	22

5	Other Interactive Global Illumination Algorithms	27
5.1	Cascaded Light Propagation Volumes	27
5.2	Screen-Space Illumination Approximations	28
5.2.1	Screen-Space Ambient Occlusion	28
5.2.2	Screen-Space Directional Occlusion	30
6	Problem Analysis	31
6.1	Analysis of the Assignment	31
6.2	Algorithm Selection	32
6.3	Framework Selection	33
7	Used Technologies and Libraries	35
7.1	GPU Computing and CUDA	35
7.1.1	Today's GPU architecture	35
7.1.2	Sample Code	37
7.2	OpenGL and GLSL	37
7.2.1	History of OpenGL and GLSL	38
7.2.2	Modern OpenGL Pipeline	39
7.2.3	OpenGL Shading Language	39
7.2.4	GPU Accelerated Tessellation	40
7.2.5	Sample Code	41
7.3	CUDA and OpenGL Interoperability	42
7.4	Another Supporting Libraries	42
7.4.1	Qt Libraries	42
7.4.2	DevIL	43
7.4.3	Open Asset Import Library	43
7.4.4	GLEW	44
8	Implementation	45
8.1	Application Structure	45
8.2	Code Structure	46
8.2.1	Resource Loading and Management	46
8.2.2	Scene Management	47
8.2.3	Rendering Subsystem	47
8.2.4	Path-Tracer	47
8.2.5	Graphical User Interface	48
8.2.6	Supporting Code and Dependencies	48
8.3	Rendering Pipeline	49
8.3.1	Preprocessing	50
8.3.2	Geometry Buffer Stage	50
8.3.3	Shadow Map Generation	51
8.3.4	Reflective Shadow Map Generation	51
8.3.5	VPL Creation Stage	51
8.3.6	ISM Splatting	53
8.3.7	Discontinuity Detection	54
8.3.8	Indirect Illumination Stage	54

8.3.8.1	Sub-Buffer Generation Stage	55
8.3.8.2	VPL Sampling	55
8.3.8.3	Indirect Illumination Filtering	56
8.3.9	SSAO Stage	57
8.3.10	Direct Illumination and Final Composition	57
8.3.11	Time Measurement	57
9	Results	59
9.1	Tested Scenes	59
9.2	Used Hardware Setups	60
9.3	Running Time Analysis	60
9.4	Quality Analysis	63
10	Conclusion	71
10.1	Summary	71
10.2	Future work	72
A	List of Abbreviations	79
B	Image Gallery	81
C	Installation and User Manual	85
C.1	Build Instructions	85
C.1.1	Console Build on Unix-Like Platforms	85
C.1.2	Using CMake-GUI	85
C.1.3	Build on Windows Platforms	86
C.2	Usage	87
D	Contents of Attached CD	89

List of Figures

1.1	The result of a global illumination algorithm (c) is usually created as a sum of the direct illumination part (a) and the indirect illumination part (b). . . .	1
2.1	BRDF is defined as a fraction of outgoing radiance L_o and incoming radiance L_i times $\cos\theta_i$	6
2.2	Direct illumination (red path) and indirect illumination (blue path)	7
2.3	256 Two-dimensional quasi random samples from Halton sequence	8
3.1	A scene rendered from the view of the light L and the camera C . Red dots indicate light depth buffer samples, blue dots indicate pixels visible from the camera C	10
3.2	Various bias settings for shadow map comparison tested on the Sponza scene. Correct bias (a), bias set too low (b) - self shadowing artifacts occur, and bias set too high (c) - shadows "flow" away from occluders.	10
3.3	Output from the geometry stage: depth buffer (a), albedo (b) and decoded normals (c)	12
3.4	Viewing frustum culling; a scene with five mesh objects with their bounding boxes, camera C and its viewing frustum VF ; boxes highlighted with a green color are intersecting the viewing frustum and are accepted. Objects highlighted with a red color are culled.	13
4.1	Four virtual point lights created on the surface after the first light bounce . .	16
4.2	A scene rendered using the instant radiosity algorithms with parameters: $N = 10, M = 20$ (a), $N = 32, M = 72$ (b), $N = 64, M = 147$ (c); courtesy of Keller [Kel97]	16
4.3	Reflective shadow map components: dept (a), world-space coordinates (b), normal (c) and flux (d), and the rendered image (e); courtesy of Dachsbacher et al. [DS05]	17
4.4	Splatting Indirect Illumination in steps; courtesy of Dachsbacher et al. [DS06]	18
4.5	The Incremental Radiosity steps; in the first step, classic VPLs are created. In the step (b) the scene is illuminated by VPLs created in the step (a). In the next frame camera and light is moved (c). The algorithm reuses some of the VPLs previously created in the step (a). Courtesy of Laine et al. [LSK ⁺ 07]	18
4.6	Discretization of viewing directions (a); the space-filling curve is used to order shadow maps (b). Courtesy of Ritschel et al. [RGKM07]	19

4.7	The average depth value Z_{avg} used for compression; courtesy of Ritschel et al. [RGKM07]	20
4.8	Zig-zag and spiral traversal in an atlas (a); the back-projected path (b); courtesy of Ritschel et al. [RGKS08]	20
4.9	A scene with reference points (\times) and entry points (\bullet); courtesy of Szécsi et al. [SSKS06]	22
4.10	A scene with two virtual lights and associated ISMs (a); the top and bottom ISM textures (b) show shadow maps without and with pull-push phase respectively. Courtesy of Ritschel et al. [RGK ⁺ 08]	23
4.11	Normalized RMS image error of the Sponza scene; the ISM resolution is varied as well as the number of point samples; courtesy of Ritschel et al. [RGK ⁺ 08]	24
4.12	Bidirectional Reflective Shadow Maps; courtesy of Ritschel et al. [REH ⁺ 11]	24
4.13	Comparison of various VPL distributions; the classic instant radiosity (2) places samples around the light source. The Bidirectional Instant Radiosity produces much better VPL distribution. The VAISM algorithm produces worse VPL distribution than BDIR, but can be evaluated more effectively. Courtesy of Ritschel et al. [REH ⁺ 11]	25
5.1	The "Crytek Sponza" scene rendered using LPV (a) (b); the image (c) shows a simple participating media scattering. Courtesy of Kaplanyan et al. [KD10]	27
5.2	The propagation scheme and the flux estimation for adjacent cells in LPV; courtesy of Kaplanyan et al. [KD10]	28
5.3	Computation of SSAO for the point P (green) with two sample rays r_1 and r_2 .	29
5.4	Ambient occlusion in the Sibenik Cathedral scene computed on-line using the screen space approach; the image was rendered in the resolution of 1280×720 pixels in 1.39 ms.	29
5.5	Comparison of image without AO (a), standard SSAO (b), SSDO (c) and SSDO with a diffuse bounce (d); the second row shows the details of SSAO (e), SSDO (f) and (g), and SSDO (h). Courtesy of Ritschel et al. [RGS09]	30
7.1	Comparison of typical CPU architecture (left) to the GPU architecture (right); courtesy of nVidia [NVI10]	35
7.2	One Fermi Streaming Multiprocessor; courtesy of nVidia [NVI09]	36
7.3	Simple parallel vector addition; an example from [NVI10]	37
7.4	Simplified OpenGL pipeline; the programmable stages are shown as light blue rectangles. Dashed rectangles denote optional stages.	39
7.5	Triangle tessellation as done by OpenGL; the triangle (a) has the tessellation level of five, the triangle (b) has the level of four. Barycentric coordinates are written next to their corresponding vertices. Courtesy of The Khronos Group [SA11]	40
7.6	A simple OpenGL drawing loop	41
7.7	A simple GLSL vertex shader	41
7.8	Simple Qt application which creates a button with the "Hello World!" sign.	42
7.9	Sample code that uses DevIL library to load an image from a file; if the operation succeeds, the width and height of the image are retrieved.	43
7.10	Loading a scene with Assimp	43

7.11	Code snippet showing GLEW usage	44
8.1	The application is designed using the Model-View-Controller (MVC) pattern. Blue modules work as the model, user input controls the scene and the rendering, green modules work as the view subsystem.	46
8.2	Example usage of a view frustum query in our implementation	47
8.3	The architecture of our ISM implementation	49
8.4	Random placement of samples using binary search in the cumulative distribution function; barycentric coordinates are generated using a method presented by Glassner [Gla93].	50
8.5	We store each VPL in a 4×4 floating point matrix. The intensity vector I is stored in the fourth row of the matrix.	52
8.6	The Crytek-Sponza scene rendered with our implementation (left) and the discontinuity buffer calculated for the same view (right)	54
8.7	Normal (a) and depth (b) sub-buffers organized in a 8×8 pattern	55
8.8	Unfiltered (a) and filtered (b) irradiance	56
8.9	Part of the Crytek-Sponza scene illuminated only by indirect illumination; an image rendered without (a) and with (b) the SSAO; the difference is especially notable around the corners.	57
9.1	Indirect illumination of the Crytek-Sponza scene; both images were rendered with a close light source position, the number of VPLs was set to 16. The low number of VPLs caused a strong flickering in the illumination. The image (c) shows the difference between the image (a) and (b).	63
9.2	Two rendered images of the Crytek-Sponza scene with a close light source positions; the number of VPLs was set to 1024. The difference in the indirect illumination is only small in contrast to the results shown in the figure 9.1. This figure shows the correct behaviour.	64
9.3	Comparison of two images of the U-shaped scene with various number of point samples used; the camera look is oriented at the part of the scene that is completely shadowed. The light is placed behind the wall. The image (a) is rendered using only 10^5 samples, the image (b) is rendered using 10^6 point samples. The light "leaks" through the wall since the low number of samples creates holes in the ISM.	65
9.4	The U-shaped scene rendered with the same settings as in the figure 9.3 (b), and the same view with the pull-push phase disabled (a); the difference is significant since holes in the ISM are not filled.	65
9.5	One ISM that was created during the Sibenik Cathedral scene rendering process. The image (a) shows the ISM without the pull-push phase, the image (b) shows the same ISM with the pull-push phase enabled. The ISM phase took 3.6 ms without the pull-push phase and 6.8 ms with the pull-push phase.	66
9.6	The image of the Monkey Box scene rendered using our ISM implementation (a) compared to the reference image, produced by our path-tracer (b); the path-length was set to match the ISM possibilities, so it terminates the tracing after the second bounce.	66

9.7	The output of our ISM implementation (a) compared to the reference image (b) of the U-shaped scene; the ISM renderer produces much brighter image, as the light "leaks" through the wall.	67
9.8	The indirect illumination rendered for all tested scenes; the number of VPLs was varied from 4 to 1024. Individual shadows are distinguishable when the VPL count is set too low. Hundreds of VPLs are usually needed to produce an acceptable result.	68
9.9	The number of point samples varied from 10^3 to 10^6 ; larger scenes require a higher count of point samples, otherwise the indirect shadows are lost. The images show only the indirect illumination.	69
B.1	The Conference Room scene rendered with our ISM renderer (a), our reference path-tracer (b) and the difference image (c); the images were rendered using the direct illumination and the first diffuse bounce of indirect illumination. . .	81
B.2	The Monkey Box scene rendered with our ISM renderer (a), our reference path-tracer (b) and the difference image (c); the images were rendered using the direct illumination and the first diffuse bounce of indirect illumination. . .	81
B.3	The Crytek Sponza scene rendered with our ISM algorithm (top) and without the indirect illumination (bottom)	82
B.4	The Sibenik scene rendered with our ISM algorithm (top) and without the indirect illumination (bottom)	83
B.5	The Crytek Sponza scene passage lit mostly by the indirect illumination . . .	84
C.1	The cmake-gui application	86
C.2	Four tabs of the tool window of our application	88

List of Tables

3.1	Depth, normal and material properties stored in three four-byte render targets	12
7.1	Official OpenGL releases with some of the features they introduced	38
8.1	Components enclosed by the core library	46
8.2	All library dependencies	48
9.1	Tested scenes with the number of triangles, geometrical and lighting complexities	59
9.2	Tested scenes with the number of triangles, geometrical and lighting complexities	60
9.3	Used hardware setups	60
9.4	Breakdown of the rendering time for all tested scenes for all hardware setups; the table shows frames per second (FPS), direct illumination and composite time (D+Mix), indirect illumination filtering time (I.Flt), indirect illumination sampling time (II), imperfect shadow map creation time (ISM), camera geometry buffer time (Cam.G), shadow map generation time (SM), screen space ambient occlusion time (SSAO), VPL generation time (VPLS) and geometry sub-buffer creation time (Sub-B). All times are in milliseconds. All scenes were rendered in the resolution of 512×512 pixels.	61
9.5	Breakdown of the rendering time for all hardware setups; all times are in milliseconds. The resolution was magnified to 1280×720 pixels. The legend is the same as for the table 9.4.	62
9.6	Comparison between the point sample generation methods; for each hardware setup we measure the time of the ISM stage when using pre-processed point samples T_s and dynamically generated point samples T_d . The speedup S shows how many times is the dynamic method faster.	62
9.7	Different geometry sub-buffer configurations and the running time of the indirect illumination stage; all times are in milliseconds.	63
9.8	Comparison between the splatting time of points T_{pt} and quads T_q ; the speedup S shows how many times is the quad splatting method slower.	63

Chapter 1

Introduction

Designers of algorithms for realistic image synthesis are usually confronted with a difficult task — the lighting evaluation. The physical principle of lighting is hard to simulate precisely, hence we are forced to create simplified lighting models. The family of global illumination algorithms tries to simulate the behaviour of lighting by using advanced lighting models, that are relatively close to the physical reality.

When we speak about global illumination in computer graphics, we often distinguish between direct illumination and indirect illumination. Direct illumination appears only on objects which are directly visible from a light source. Indirect illumination emerges from the repetitious light bounces. Global illumination algorithms combine direct illumination with indirect to maximize the realism of produced synthetic images.

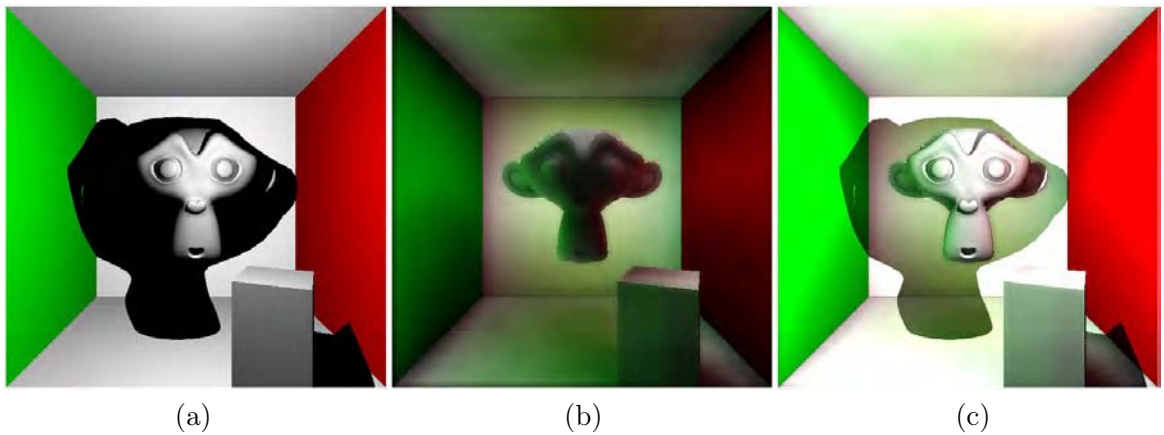


Figure 1.1: The result of a global illumination algorithm (c) is usually created as a sum of the direct illumination part (a) and the indirect illumination part (b).

An example of global illumination breakdown is shown in the figure 1.1. The image (a) was rendered considering only the direct illumination. The image (b) contains only the indirect illumination and the image (c) is the sum of both parts. As can be seen in the image (a), the areas that are not directly visible from the light source appear completely dark. Many natural lighting effects cannot be simulated without the global illumination

algorithms. One example is the color bleeding. The head of the monkey in the figure 1.1 gets imbued by the red and green walls as the bounced light carries the color of the surface.

1.1 Motivation

The direct illumination is naturally easier to evaluate, since it considers only one light bounce. That typically involves only two visibility tests. The direct illumination is usually the necessity that has to be calculated if we want to synthesize images. However, if we want to generate physically more accurate image, we have to evaluate the indirect illumination part as well [Sto04].

Despite its importance, the indirect illumination is often omitted or very coarsely approximated in many real-time or interactive applications. The example of a commonly used coarse indirect illumination approximation can be found in the Phong reflection model [Pho75], which uses an ambient term to express the indirect illumination. The problem of a correct global illumination evaluation is its computational complexity. In contrast to the direct illumination, a global illumination algorithm has to trace the light on its path, which typically consists of many bounces. That involves the evaluation of many visibility checks.

As long as the scene remains unchanged, the light paths can be precomputed and interactive frame rates can be achieved [SSKS06]. However, if the scene is completely dynamic, we cannot simply use the precomputed paths or visibilities, and the current affordable hardware does not allow us to precisely evaluate all the light paths if we want to keep the real-time performance. Stochastic algorithms are the well-known "solution" of this problem. Monte Carlo and Quasi-Monte Carlo estimators have a long history in the computer graphics algorithms, and can be also used to evaluate the indirect illumination [Laf96][PH04].

1.2 Subject of this Thesis

This thesis investigates the Instant Radiosity method, introduced by Keller in 1997 [Kel97], and research done around the Virtual Point Light (VPL) based algorithms. VPL based algorithms usually produce a plausible global illumination approximation with a budget of only a few milliseconds. We also implement the Imperfect Shadow Maps (ISM) algorithm, introduced by Ritschel et al. in 2008 [RGK⁺08]. The implementation is designed to take a benefit of a modern GPU hardware.

Another algorithms used for realistic image synthesis are, for instance, photon mapping [Jen01] or path tracing [Kaj86]. In contrast to those algorithms, Instant Radiosity and similar approaches are mainly targeted to the real-time or interactive uses, so they typically produce less accurate images. We implement a simple path tracing algorithm as a reference solution and compare its output to our ISM based renderer. As discussed later, VPL based algorithms make a really good trade-off between the render quality and the computational time.

1.3 Thesis Structure

We divide our thesis into ten chapters. The chapter 2 briefly recapitulates the theoretical background of the realistic image synthesis. The thesis continues in the chapter 3, where we describe the common rasterization techniques, since our implementation is based on the rasterization. We review the existing Instant Radiosity methods in the chapter 4. Rendering algorithms reviewed in the chapter 5 are not based on Instant Radiosity, but they are also treated as a real-time global illumination solution. Our assignment is further analysed in the chapter 6, where we discuss the choice of the ISM algorithm for our implementation. The chapter 7 briefly describes the software and hardware technologies we have used in our application. We go through our implementation details in the chapter 8. Our results are presented in the chapter 9. We sum up the contribution of our thesis in the last chapter 10. The user manual, detailed code structure and image gallery are enclosed in appendices.

Chapter 2

Theory behind the Radiosity Methods

In this chapter we briefly recapitulate the fundamental theory of the realistic image synthesis. In following sections we describe radiometric quantities and the bidirectional reflectance distribution function. We clarify the terms Global Illumination and Rendering Equation. We also describe the principle of the radiosity methods and the Monte Carlo integral estimation.

2.1 Scene Representation

The majority of reviewed algorithms can deal only with the boundary representation of the scene geometry. Also most of the described algorithms use the triangle representation. Other representations, like CSG or volumetric data, have to be converted to triangles in a preliminary stage. Participating media, such as fog, are usually ignored by the reviewed algorithms.

2.2 Important Radiometric Quantities

In this section we briefly describe basic radiometric quantities which we use in this thesis. The radiant energy is measured in joules (J). The common symbol for radiant energy is Q . Radiant flux expresses the flow of the radiant energy Q in the time t .

$$\Phi = \frac{dQ}{dt} \quad [W, J, s] \quad (2.1)$$

The symbol Φ denotes the radiant flux, the unit of the radiant flux is watt (W). The most frequently used quantity in this thesis is the radiosity. Radiosity is defined as the radiant flux Φ coming through the area A .

$$E(x) = \frac{d\Phi(x)}{dA} \quad [Wm^{-2}, W, m^2] \quad (2.2)$$

The radiance L expresses the radiant flux Φ per solid angle per area A .

$$L(x, \omega) = \frac{d\Phi(x)}{\cos\theta dA d\omega} \quad [Wm^{-2}sr^{-1}, W, m^2, sr] \quad (2.3)$$

The angle θ is measured between the normal of the area A and the direction ω , from where the radiance is coming.

2.3 Bidirectional Reflectance Distribution Function

As we deal with the boundary representation, we have to describe the surface material properties. The Bidirectional Reflectance Distribution Function (BRDF) expresses the fraction of the radiance L_o outgoing from the point x when it is lit by the incoming radiance L_i . For a single light wavelength, the surface reflectance is defined by the equation (2.4).

$$f_r(\omega_i, x, \omega_o) = \frac{dL_o(x, \omega_o)}{dE(x, \omega_i)} = \frac{dL_o(x, \omega_o)}{L_i(x, \omega_i) \cos \theta_i d\omega_i} \quad (2.4)$$

The figure 2.1 shows the meaning of variables in the definition (2.4).

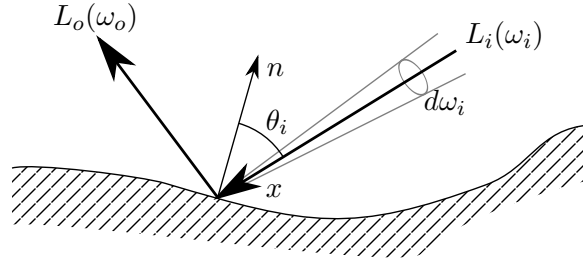


Figure 2.1: BRDF is defined as a fraction of outgoing radiance L_o and incoming radiance L_i times $\cos \theta_i$.

In a free space, the radiance L_i incoming from the direction ω is equal to the outgoing radiance L_o in the direction $-\omega$. We can write $L_i(x, \omega) = L_o(x, -\omega)$ for every point x in the free space.

2.4 Global Illumination

The fundamental equation which expresses the energy transportation within the scene was described by Kajiya in 1986 and is called radiance equation [Kaj86].

$$L(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \int_{\Omega} f_r(\vec{\omega}_i, x, \vec{\omega}_r) L(h(x, \vec{\omega}_i), -\vec{\omega}_i) \cos \theta_i d\omega_i \quad (2.5)$$

The radiance equation (2.5) shows that the radiance L coming from the scene point x in the direction $\vec{\omega}_r$, is given by a sum of two terms. The radiance emitted from the point x in the direction $\vec{\omega}_r$ is expressed by the L_e term. The integral over the hemisphere Ω sums the reflected radiance. For each incoming direction $\vec{\omega}_i$, the reflected radiance can be computed as a product of the BRDF f_r and the incoming radiance $L(h(x, \vec{\omega}_i), -\vec{\omega}_i) \cos \theta_i$, where h is the ray shooting function, which finds the closest scene point to the x in the direction $\vec{\omega}_i$. The $\cos \theta_i$ projects the incoming radiance direction onto the surface normal vector. The radiance equation (2.5) holds only if the light is transported through optically inactive medium like vacuum or dry air.

The figure 2.2 shows two examples of illumination types that we distinguish for points that are visible from the camera. The red path denotes the direct illumination, since it leads directly from the light source, bouncing off the surface and then to the camera. The blue path adds one more light bounce. The surface is then lit indirectly, we call global illumination algorithms combine both direct and indirect illumination.

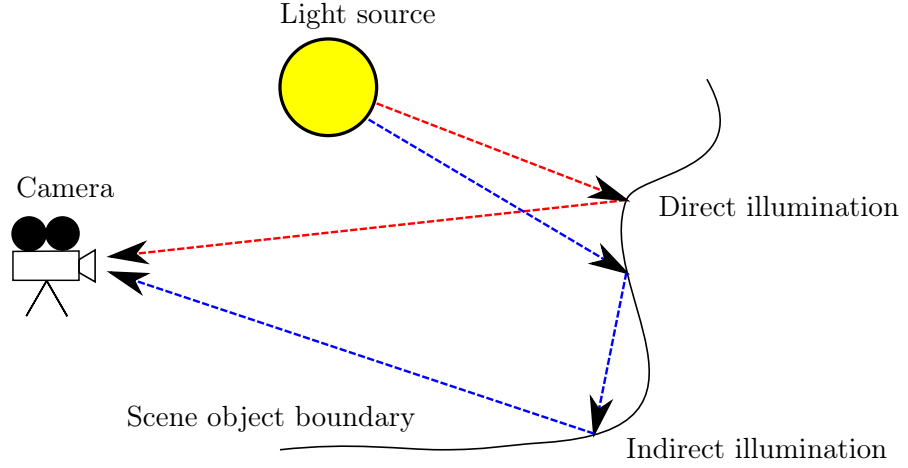


Figure 2.2: Direct illumination (red path) and indirect illumination (blue path)

2.5 Radiosity Methods

Assuming a constant BRDF, a.k.a. the ideal diffuse surface, the f_r term can be substituted with $\frac{\rho(x)}{\pi}$ and the radiance equation can be simplified to the form shown in (2.6).

$$L(x) = L_e(x) + \frac{\rho(x)}{\pi} \int_{\Omega} L(h(x, \vec{\omega}_i)) \cos \theta_i d\omega_i, \quad (2.6)$$

where $\rho(x)$ expresses the diffuse reflectivity (also known as albedo) of the scene point x . Rewriting the equation to integrate over all scene elements (surfaces) instead of angles we receive:

$$L(x) = L_e(x) + \frac{\rho(x)}{\pi} \int_S L(y, y \rightarrow x) G(x, y) V(x, y) dA, \quad (2.7)$$

where $L(y, y \rightarrow x)$ expresses the radiance coming from the scene point y to the point x , the geometric factor $G(x, y)$ expresses the mutual pose of the point x and y .

$$G(x, y) = \frac{\cos \theta_x \cos \theta_y}{\|x - y\|^2} \quad (2.8)$$

Visibility from the point x to the point y is expressed by the visibility factor $V(x, y)$, which is equal to either 0 or 1. As the radiance outgoing from the scene point x is independent of the direction $\vec{\omega}_r$ and equal to radiosity of the point x divided by π , the radiance equation can be multiplied by π and rewritten to express the radiosity:

$$B(x) = B_e(x) + \rho(x) \int_S B(y) \frac{G(x, y) V(x, y)}{\pi} dA \quad (2.9)$$

If we deal with a scene which is made of a finite number of diffuse surfaces we can substitute the integral with the sum:

$$B(x) = B_e(x) + \rho(x) \sum_{j=1}^N \int_{A_j} B_j \frac{G(x, y) V(x, y)}{\pi} dA_j \quad (2.10)$$

The mean radiosity of the i -th scene element B_i can be computed as:

$$B_i = \frac{1}{A_i} \int_{A_i} B(x) dA_i = B_{e,i} + \rho_i \sum_{j=1}^N B_j \int_{A_i} \int_{A_j} \frac{G(x,y)V(x,y)}{\pi} dA_j dA_i \quad (2.11)$$

Introducing the form factor $F_{i,j} = \int_{A_i} \int_{A_j} \frac{G(x,y)V(x,y)}{\pi} dA_j dA_i$ we can write the classic radiosity equation [GCT86]:

$$B_i = B_{e,i} + \rho_i \sum_{j=1}^N B_j F_{i,j} \quad (2.12)$$

2.6 Quasi-Monte Carlo Estimation

In many cases, reviewed algorithms use the Quasi-Monte Carlo integration [Laf96]. Monte Carlo methods estimate the value of integrated function by summing a finite number of samples:

$$\int_{I^s} f(x) dx \approx \frac{1}{N} \sum_{i=1}^N f(x_i), \quad (2.13)$$

where I^s is an s -dimensional interval and $x_i \in I^s$.

The example of a quasi-random sequence is shown in the figure 2.3. Halton sequence is one of the favourite sources of quasi-random multidimensional vectors [Hal64]. It gained popularity for its low discrepancy and relatively straight-forward implementation.

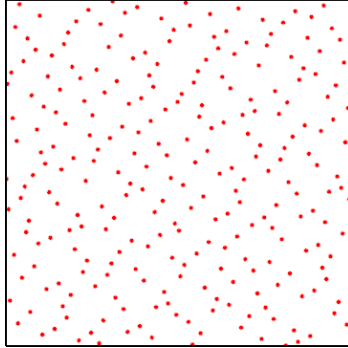


Figure 2.3: 256 Two-dimensional quasi random samples from Halton sequence

Chapter 3

Common Rasterization Techniques

In this chapter we briefly recapitulate some of the well-known rasterization techniques, which are important for our implementation. We explain the basic depth-buffer based shadow mapping. We also briefly discuss deferred rendering techniques and the idea of view frustum culling.

3.1 Shadow Mapping

Shadow mapping is a common technique for shadow generation. It uses the fact that only pixels which are visible from the light source are directly illuminated. Pixels not seen by the light are in direct shadow. The algorithm uses a depth buffer rendered from the light view to compute the visibility. When the light depth buffer texture is used for the shadow generation, it is often called shadow map.

3.1.1 Point Light Sources

The principle of basic shadow mapping, which works only with point light sources, is shown in the figure 3.1. The depth buffer samples as seen from the light are shown as red dots (in this case the shadow map has the resolution of six pixels). When the direct illumination of camera samples (blue dots) is evaluated, the pixels are translated and projected as they would be seen by the light source L . For each camera view sample corresponding shadow map texel is fetched and the depth value is compared to the light view depth of a current camera view sample. In continuous domain only two scenarios can occur.

1. Depth stored in the shadow map is **equal to** the light view depth of a current sample.
2. Depth stored in the shadow map is **lesser than** the light view depth of a current sample.

The current sample is visible from the light source in the first case. As both samples have the same depth and light space coordinates, they were taken from the same point in the scene. But when the depth stored in the shadow map is lesser than the light view depth of a current sample, the current sample is occluded (there was another point in the scene closer

to the light). This scenario is shown in the figure 3.1 where the light view sample LS_2 is closer to the light than the camera view sample CS_1 .

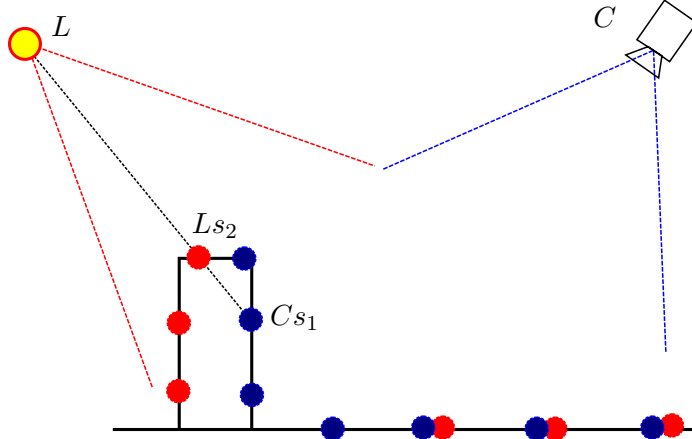


Figure 3.1: A scene rendered from the view of the light L and the camera C . Red dots indicate light depth buffer samples, blue dots indicate pixels visible from the camera C .

However, in the real application two major problems can occur. In the first place, the shadow map is stored in a buffer/texture which has not unlimited numerical precision. So we cannot rely on *equal to* operator during the shadow map comparison, because it would cause artifacts due to limited numerical precision. Secondly the shadow map is stored as a discretization of continuous world. This has to be taken into account to avoid sampling artifacts.

Both problems can be partially solved using small bias which is added to the shadow map samples. As shown in the figure 3.2 biasing can cause another artifacts, so the bias value has to be carefully chosen to fit desired quality requirements.

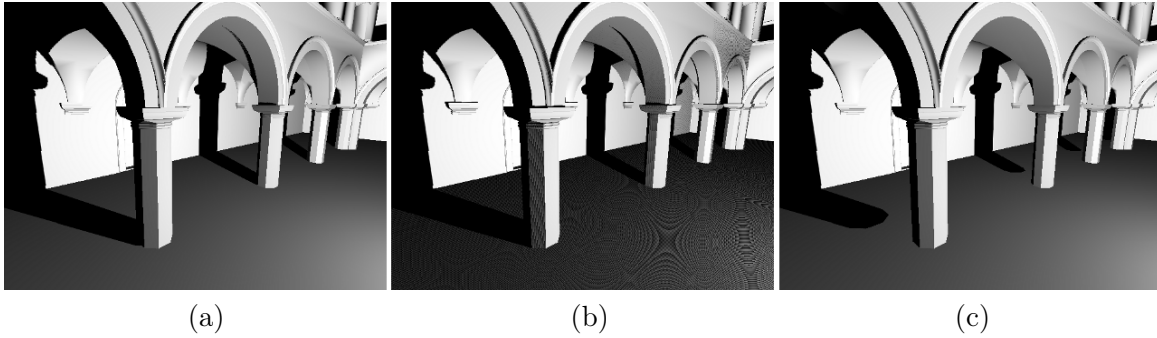


Figure 3.2: Various bias settings for shadow map comparison tested on the Sponza scene. Correct bias (a), bias set too low (b) - self shadowing artifacts occur, and bias set too high (c) - shadows "flow" away from occluders.

Another pitfall of shadow mapping is the limited resolution of depth buffer. If the shadow map resolution is set too low, the shapes of actual shadow map pixels can appear in the image.

Blurring the shadow can remove those artifacts and create more pleasant results, but the created soft shadow is physically incorrect.

The great advantage of shadow mapping is its straight-forward implementation. Only the mutual transformation of viewer and light is required. The depth buffer can be rendered using common rasterization engines, such as OpenGL. Shadow mapping requires no scene preprocessing, but acceleration data structure can be used to cull geometry and speedup the shadow map generation. Recent papers also present smarter culling approaches, such as Shadow Caster Culling for Efficient Shadow Mapping [BMSW11].

3.1.2 Area Light Sources

Because the basic shadow map algorithm can distinguish only lit pixels from occluded pixels, it can handle only point lights. Area light sources require an estimation of how much is the area of the light source visible from the current pixel. One approach is to sample the area of the light source in Monte Carlo fashion and create one shadow map for each light sample. Temporal coherence can be used to achieve real-time frame rates, this technique was introduced by Scherzer et al. [SSMW09].

Other known techniques require only one shadow map per light source. They estimate the occluded area by backprojection the shadow map pixels on the light source [GBP06]. Or sample neighbouring shadow map pixels and average the occlusion [Fer05].

3.2 Deferred Rendering

Deferred shading redesigns the standard forward rasterization pipeline to speed up the lighting computation in complex scenes. In a simple forward pipeline, geometry is translated to the camera coordinate system and then rasterized. During the rasterization process, lighting is evaluated (that can include complex computations). Performance problems can occur when computed pixels are frequently overdrawn, for example by closer objects.

Deferred rendering algorithms split a geometry phase from a lighting phase, so the final image is created in two or more stages. In the first "geometry" phase the whole scene is rasterized, but only geometrical and material properties are evaluated and saved. The second phase reads the geometrical properties (stored in buffers/textures) and evaluates the lighting. This approach was introduced by Deering et al. in 1988 [DWS⁺88]. The concept of geometry buffers (g-buffers) was introduced by Saito et al. in 1990 [ST90].

If the lighting evaluation creates a bottleneck in the rasterization process, it is vital to keep the amount of evaluated pixels low. Deferred rendering elegantly solves this issue by evaluating lighting only on visible pixels. If the scene contains complex geometry and it is impossible to avoid overdrawing (for example by rendering closer objects first), the deferred pipeline beats the forward pipeline as the overdrawn pixels contain only geometrical and material properties which are much cheaper to evaluate than a complex lighting. However, if there is no expensive lighting evaluation required, the forward approach is typically faster as it requires lower memory bandwidth.

Deferred rendering typically requires multiple render targets (MRT) to be available as the first stage produces more data than a four channel image. The challenge is to find

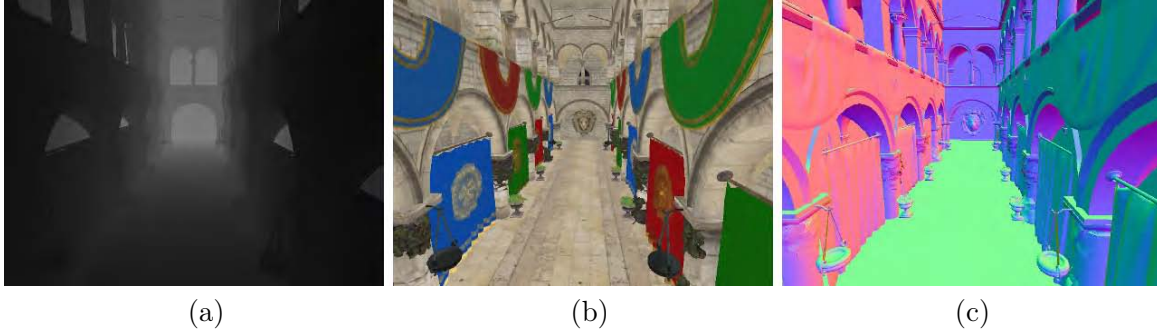


Figure 3.3: Output from the geometry stage: depth buffer (a), albedo (b) and decoded normals (c)

efficient mapping of geometrical and material properties to available buffer formats. Wasteful mapping can cause a memory bandwidth bottleneck. The example mapping is shown in the table 3.1. Depth is stored as four-byte value, normal is packed into two two-byte values using sphere-map encoding [Mit09]. The last render target contains red, green and blue color components and the specular exponent, one byte per component. The camera space position can be later recomputed from the stored depth and screen space coordinates.

	Byte			
RT	0	1	2	3
0	depth			
1	norm.x		norm.y	
2	alb.r	alb.g	alb.b	spec

Table 3.1: Depth, normal and material properties stored in three four-byte render targets

Deferred shading also simplifies the pipeline and shader design. There is no need to have huge number of shaders created by the combination of all geometry types with all lighting settings. It is also much simpler to extend existing deferred engine with new features (changing the geometry phase does not affect lighting phase and vice versa).

3.3 Geometry Culling

To reduce the number of drawn primitives per frame, rasterization engines often use some kind of geometry culling. Graphic scenes are usually made of many stand-alone geometrical objects. This scheme follows the real world situation where, for example, a room can contain separate pieces of furniture. The simple approach is to distinguish between objects necessary for the image generation and objects that do not contribute to the rendered image. If the object is unnecessary, it can be omitted from the rendering procedure and computational time can be spared.

For example, only objects that lie in the camera view, can be visible on the screen. The common practice is to use simplified representations of particular objects, such as bounding volumes, in combination with the viewing frustum intersection test. If the object bounding

box intersects with the viewing frustum, all its geometry is drawn. Otherwise, it is culled away. This technique is called Viewing Frustum Culling (VFC). A simple scene with a camera and a few objects is shown in the figure 3.4.

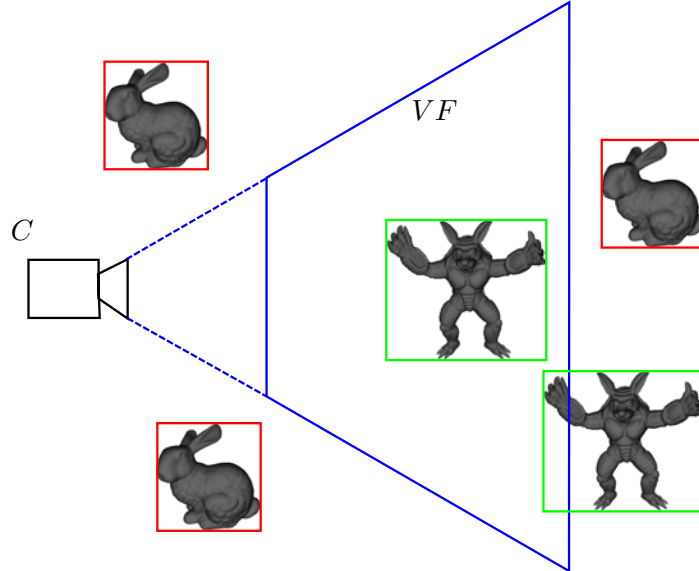


Figure 3.4: Viewing frustum culling; a scene with five mesh objects with their bounding boxes, camera C and its viewing frustum VF ; boxes highlighted with a green color are intersecting the viewing frustum and are accepted. Objects highlighted with a red color are culled.

VFC is often used with bounding volume hierarchies. There are also advanced techniques which employ hardware acceleration occlusion tests [BMW09].

Chapter 4

Family of the Instant Radiosity Algorithms

This chapter reviews the important research done around the interactive global illumination algorithms, focusing on Instant Radiosity and similar approaches. Instant Radiosity itself is described in the section 4.1. The Reflective Shadow Maps algorithm is discussed in the section 4.2. The following section 4.3 reviews the Splatting Indirect Illumination approach. Incremental Instant Radiosity is described in the section 4.4. The section 4.5 describes the Coherent Shadow Maps and the following section 4.6 the successive Coherent Surface Shadow Maps. So far known VPL sampling improvements are discussed in the section 4.7. The Precomputed Radiance Map approach is reviewed in the section 4.8. The section 4.9 describes the Implicit Visibility and Antiradiance approach. Algorithm that combines global and local virtual point lights is reviewed in the section 4.10. The last section of this chapter 4.11 reviews the Imperfect Shadow Maps algorithm.

4.1 Instant Radiosity

The Instant Radiosity algorithm was presented by Keller in 1997 [Kel97]. He uses a quasi-random walk through the scene, creating virtual point lights (VPL) as the ray bounces the surface. Scene is rendered from each VPL storing the dept information in a shadow map. After that, scene can be lit using virtual lights in same way as ordinary point lights. Shadow maps are used to check visibility of each virtual light from the target point.

The algorithm starts with N light particles. Halton sequence is used to place N light particles over the light sources in the scene. Particles are then shot to the scene in quasi-random directions. To estimate which particles should be absorbed by the surface and which continue bouncing, the mean scene reflectivity is calculated:

$$\bar{\rho} = \frac{\sum_{k=1}^K \rho_{d,k} |A_k|}{\sum_{k=1}^K |A_k|}, \quad (4.1)$$

where K stands for the number of scene elements A_k with average diffuse reflectivity $\rho_{d,k}$. The mean reflectivity $\bar{\rho}$ is used in fractional absorption. After first bounce $\bar{\rho}N$ particles

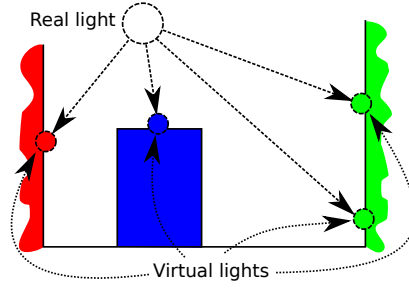


Figure 4.1: Four virtual point lights created on the surface after the first light bounce

should remain, $\bar{\rho}^2 N$ after second bounce and so on. The number of radiance points M generated by the random walk is given by the equation (4.2).

$$M < \sum_{i=0}^{\infty} \bar{\rho}^i N = \frac{1}{1 - \bar{\rho}} N \quad (4.2)$$

The equation requires $0 \leq \bar{\rho} < 1$ which holds in all real scenes. The number of radiance samples M is linear in N . The scene is lit by all those generated VPLs. The first implementation uses multiple render passes and accumulation buffer to sum the influences of all generated lights together.

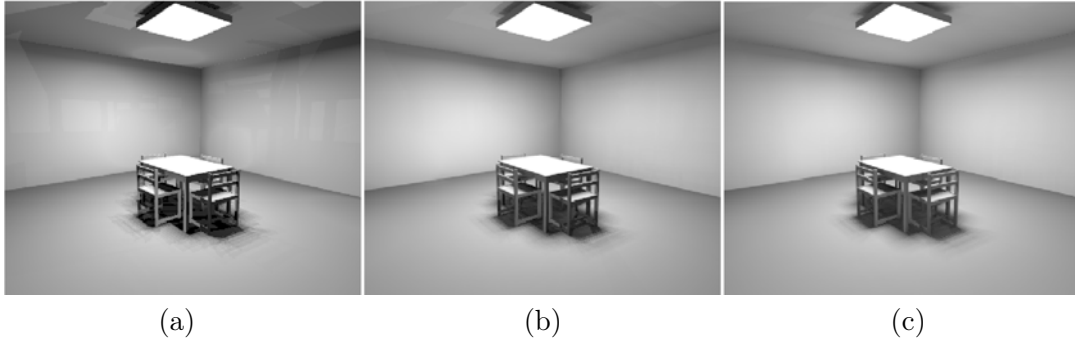


Figure 4.2: A scene rendered using the instant radiosity algorithms with parameters: $N = 10, M = 20$ (a), $N = 32, M = 72$ (b), $N = 64, M = 147$ (c); courtesy of Keller [Kel97]

In the figure 4.2 you can see various settings for the instant radiosity algorithm. The low number of light samples can cause disturbing artifacts (the leftmost image). The images show that for a smaller scene, tens to hundreds of virtual lights should be sufficient.

The great advantage of instant radiosity is that it can work without any acceleration data structures, so it can be easily used on fully dynamic scenes. The first implementation from Keller uses hardware accelerated rasterisation. But the idea can also be used with other rendering techniques like ray-tracing.

4.2 Reflective Shadow Maps

This technique extends classic shadow-maps and creates virtual light in every shadow-map pixel [DS05]. The additional information in shadow-map is needed, so this algorithm requires multiple render targets to be available.

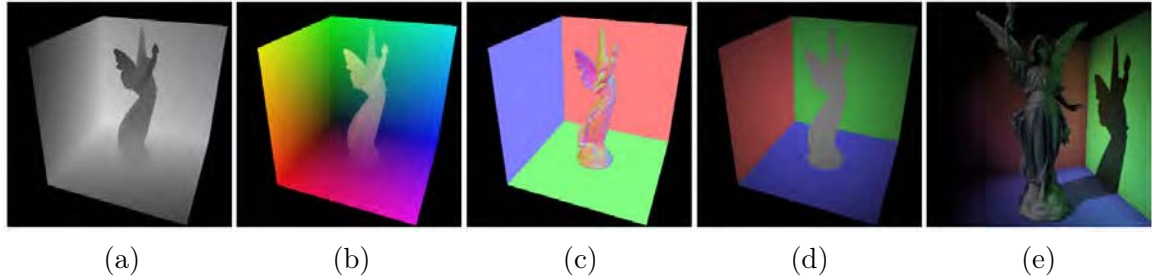


Figure 4.3: Reflective shadow map components: dept (a), world-space coordinates (b), normal (c) and flux (d), and the rendered image (e); courtesy of Dachsbacher et al. [DS05]

The algorithm first renders the scene from the light view, storing world-space position, normal and flux for each pixel. When pixels seen by a camera are processed, the reflective shadow map (RSM) pixels are considered as a source of indirect illumination. To avoid evaluation from all RSM pixels (shadow-map can contain hundreds of thousands of pixels), only a random subset of RSM pixels is used for one pixel on the screen. The sampling assumes that if two points in the scene are close to each other, their projections on the screen will be close to each other. This assumption cannot be used in the opposite manner. But as the sampling does not need to be accurate, RSM samples are selected close to currently evaluated pixel. Also the visibility check between the pixels and RSM samples is omitted. This simplification can lead to totally wrong results.

Another speedup can be achieved when interpolating the indirect illumination where it does not vary much. The algorithm first renders the scene (from the camera view) in low resolution and evaluates the indirect illumination for all pixels. In the second pass the scene is rendered in a full resolution, and where it is possible, the indirect illumination is interpolated from the low resolution image. The variation of indirect illumination is guessed by comparison the high resolution pixel normal and the low resolution pixel normal. Pixels which have similar normal in low and high resolution buffers use the interpolated indirect illumination.

4.3 Splatting Indirect Illumination

The research published by Dachsbacher et al. in 2006 [DS06] use the idea of RSM [DS05] and creates a light source in every shadow-map pixel. But the contribution of pixel light sources to the final camera image is not computed per camera image pixel. The approach is reversed and for each group of virtual pixel lights the indirect illumination is splatted into its screen space neighborhood. The shape of the light splat can be modified to approximate BRDF. This can be used to create simple reflection and refraction effects like caustics.

This algorithm belongs to a deferred shading family by its nature. So the scene is rendered using multiple render targets storing the world-space position, normal and material properties for each pixel. The direct illumination and also the splatting is done in the deferred pass. Dachsbacher et al. use importance sampling of RSM to select pixels with higher flux. The algorithm is combined with ambient occlusion to approximate local visibility effects like self-shadowing.

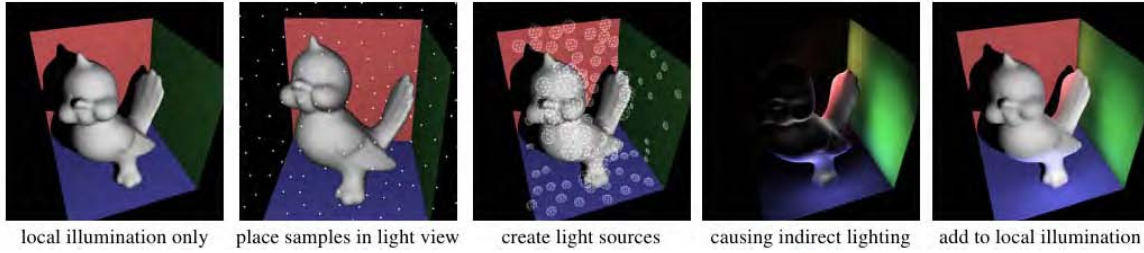


Figure 4.4: Splatting Indirect Illumination in steps; courtesy of Dachsbacher et al. [DS06]

With the use of RSM everything can be done on GPU using rasterisation pipeline. The splatting of indirect light can also be used together with another VPL based algorithm.

4.4 Incremental Instant Radiosity

The Incremental Instant Radiosity algorithm was presented by Laine et al. in 2007 [LSK⁺07]. The algorithm is based on classic instant radiosity method [Kel97], but it does not create all the VPLs from scratch every frame. It spares time by reusing valid VPLs and their shadow-maps from previous frames. The algorithm also iteratively improves VPL distribution over the scene.

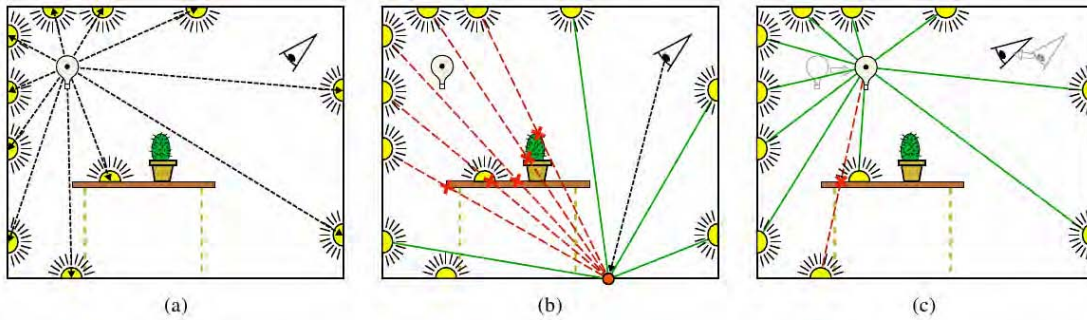


Figure 4.5: The Incremental Radiosity steps; in the first step, classic VPLs are created. In the step (b) the scene is illuminated by VPLs created in the step (a). In the next frame camera and light is moved (c). The algorithm reuses some of the VPLs previously created in the step (a). Courtesy of Laine et al. [LSK⁺07]

In the beginning VPLs are created the same way as in the classic instant radiosity method. In all other frames the algorithm detects invalid VPLs, deletes them and creates new ones.

The VPL can become invalid if the light moves and (1) the VPL becomes occluded by geometry, or (2) the VPL moves out of the light illumination region. The occlusion is detected with the help of ray-tracing on CPU. For all primary light sources the algorithm maps the directions to existing VPLs to 2D domain, where it calculates Voronoi diagram. The new lights are placed in directions which corresponds to the biggest empty space in the 2D domain which reduces the dispersion. The Voronoi diagram is used to find such places. After a new direction is found, the ray is casted from the primary light source. The intersection with the ray and the scene creates a new virtual point light which implies new shadow-map generation.

If the primary light source moves only smoothly, the algorithm can reuse huge number of previously created VPLs and can bring performance benefits. However, the algorithm uses ray-tracing, so the acceleration data structure is needed. This creates limitation for dynamic scenes. The scene can be split to static and dynamic parts. The static part is included in the acceleration data structure and considered when tracing the light paths. But the dynamic part cannot influence the VPL generation. So all dynamic objects just receive global illumination.

4.5 Coherent Shadow Maps

The visibility queries are the most time consuming operations used in global illumination. Coherent Shadow Maps (CSM) [RGKM07] and Coherent Surface Shadow Maps (CSSM) [RGKS08] create a data structure which contains precomputed visibility information.

The hypothetical data structure should hold visibility information for each pair of points in the scene. For example it should contain shadow-maps for all objects rendered from some set of directions. This is however infeasible due to high storage requirements. The CSM data structure tries to solve this problem using lossless compression and exploiting the coherence between neighbouring shadow maps.

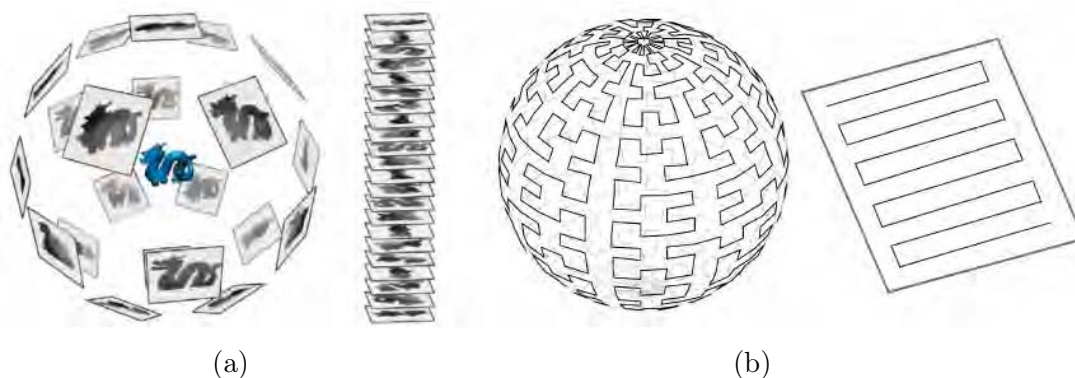


Figure 4.6: Discretization of viewing directions (a); the space-filling curve is used to order shadow maps (b). Courtesy of Ritschel et al. [RGKM07]

For each object in the scene the algorithm first discretizes the directions around the object and creates N shadow maps. The compression works best if the neighbouring shadow

maps contain similar depth values. Zig-zag or Hilbert space-filling curves are used to achieve the coherence between shadow-maps.

The idea behind used compression is shown in the figure 4.7. The classic shadow map contains only the closest depths (Z_1 in the image). But if we test visibility along the ray shown in the figure 4.7, any value between Z_1 and Z_2 will produce correct result. The graph in the figure 4.7 shows the values of Z_1 and Z_2 depending on the shadow map index i . Only three values Z_{avg} are sufficient to compress the N shadow maps.

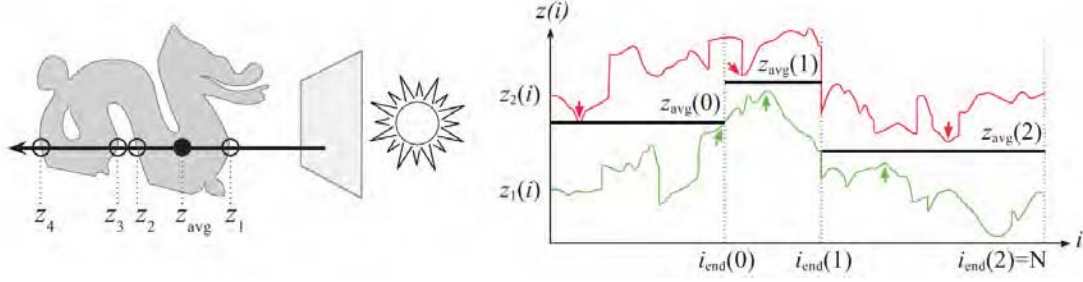


Figure 4.7: The average depth value Z_{avg} used for compression; courtesy of Ritschel et al. [RGKM07]

As the discretization creates aliasing artifacts, the CSM has to be filtered. The Coherent Shadow Maps are created per object, so to avoid reconstruction of CSM all objects must remain rigid. But any object can be moved in any degree of freedom. The main disadvantage of the CSM is that they can only be used for visibility checks from points that are outside the convex hull of the scene objects.

4.6 Coherent Surface Shadow Maps

Coherent Surface Shadow Maps (CSSM) removes the main limitation of CSM discussed in previous section [RGKS08]. To allow in-convex hull tests CSSM are created by rendering cube shadow maps from the object surface. The object surface is discretized and in each point a full cube map is rendered, storing depths to all directions.

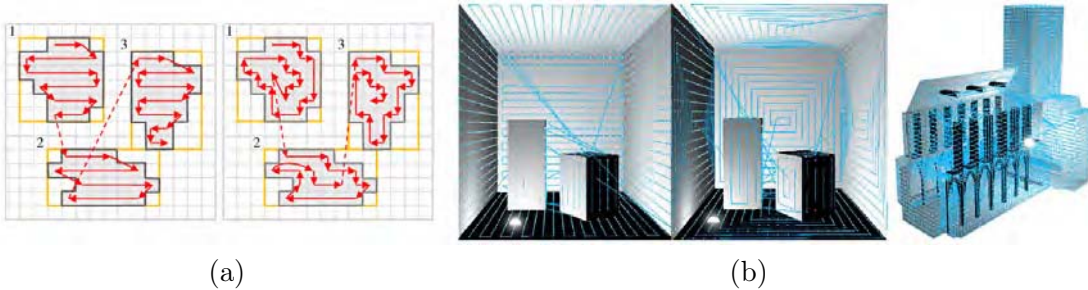


Figure 4.8: Zig-zag and spiral traversal in an atlas (a); the back-projected path (b); courtesy of Ritschel et al. [RGKS08]

The problem is to generate coherent paths over the surface. To solve this problem, a texture atlas can be used. The texture atlas contains unwrapped surface properties from all objects in one big texture. The atlas usually contains material properties, normal and position for all surface points. For the atlas generation common content creation software can be used. With texture atlas, the path can be generated in 2D in the atlas and back-projected to 3D. Two sample paths can be seen in the figure 4.8. The compression of CSSM is based on the same idea as in CSM.

4.7 VPL Sampling Improvements

Many improvements have been found in the field of virtual point light (VPL) distribution over the scene [SIMP06][REH⁺11]. The naive distribution of virtual lights does not produce optimal results, especially in scene views where the illumination comes completely from indirect bounces.

4.7.1 Simple Iterative Method

A simple iterative method has been presented by Georgiev et al. in 2010 [GS10]. The algorithm estimates the average contribution of VPLs and uses Russian roulette to discard unimportant VLPs. The estimate of contribution can be used from previous frames, so this algorithm works best with multi-pass renderers.

4.7.2 Bidirectional Instant Radiosity

Bidirectional Instant Radiosity (BDIR) [SIMP06] uses the same idea as bidirectional path tracing [LW93]. The task of BDIR is to find relevant paths from a light source to the camera. Simple observation can be used to find out that only VPLs that are closer than two bounces to the camera can cause visible illumination. So the algorithm uses inverse instant radiosity to create virtual point lights by tracing rays from camera.

First $N/2$ VPLs are created by tracing paths (of length two) from camera. The second half of VPLs is created normally by casting rays from the light source. For each of N lights, the power brought to the camera is evaluated. The cumulative distribution function is built and the set of N lights is re-sampled.

4.8 Precomputed Radiance Map

For static scenes, the radiance transfer can be precomputed. Szécsi et al. [SSKS06] use this approach. The algorithm works with two kinds of samples. The scene is covered with a number of "reference points" and "entry points". In the preprocessing stage the paths from entry points to reference points are computed and stored with the probability of the path. In the rendering stage, if an entry point receives radiance the precomputed radiance map shows to which reference points the radiance can go. The incoming radiance between reference points is interpolated.

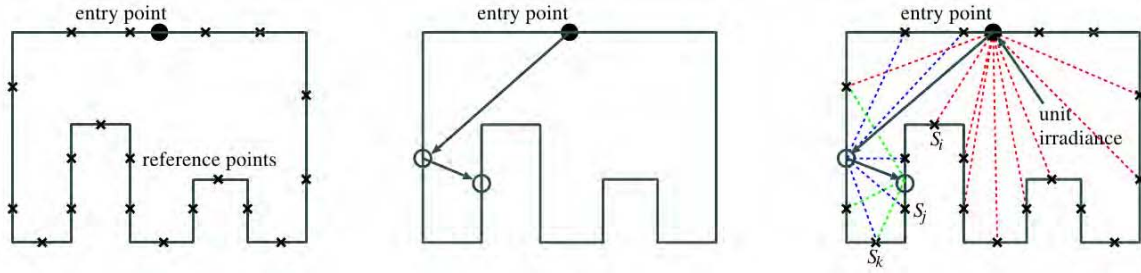


Figure 4.9: A scene with reference points (\times) and entry points (\bullet); courtesy of Szécsi et al. [SSKS06]

4.9 Implicit Visibility and Antiradiance

Dachsbacher et al. reformulate the rendering equation to contain implicit visibility [DSDD07]. The extra illumination is compensated with a new quantity called antiradiance.

The classic rendering equation [Kaj86] calculates the outgoing radiance by integrating the radiance incoming from visible objects (dimmed by BRDF). In the new approach, the visibility step is skipped and the radiance comes from all scene points. But every point also emits incident light backwards, this quantity is called antiradiance. The algorithm discretizes the scene to spacial and directional bins. The hierarchy is build to handle scene complexity.

An iterative solver, similar to Hierarchical Radiosity with Clustering [SAG94], is used. Dachsbacher et al. present an efficient GPU implementation which converges in interactive times. Moving objects and lights are handled.

4.10 Combination of Global and Local VPLs

The instant radiosity was designed to work well with diffuse surfaces. But if the scene contains glossy or specular materials, special approach has to be used. One of the possible solutions was presented by Davidovic et al. in 2010 [DKH⁺10]. Their algorithm separates global low-rank and local high-rank components.

The global lights are clustered and all lights in one cluster share one shadow map. For local lights, the visibility is ignored.

4.11 Imperfect Shadow Maps

Direct lighting often causes high frequency illumination changes. But it has been found that indirect illumination alters only smoothly in most scenes [WRC88]. In contrast to the direct illumination, which has to be precisely evaluated, indirect illumination does not necessary have to be precise [AFO05]. The most time consuming part of indirect illumination computation are visibility queries [RGKS08]. The indirect illumination evaluation can be accelerated with the use of low resolution shadow maps which store only approximate depth values [RGK⁺08]. Those Imperfect Shadow Maps (ISM) are used for visibility queries the

same way as regular shadow maps are, but they are not created from the precise scene representation.

ISM are rendered using point based scene representation which approximates the scene geometry. The point based representation is created from boundary representation in pre-processing stage. For each point the algorithm first selects one of the scene triangles with the probability proportional to the area of the triangle. Then the point is placed on a random location on the selected triangle. This leads to uniform distribution of points over the scene. ISMs can deal with fully dynamic scenes but the points approximating the scene have to be transformed with the moving geometry.

The depth information stored in ISM does not have to be precise, so each ISM is rendered using only a subset of all scene points. During the ISM render stage the stream of all scene points is split and points are randomly sent to different ISMs. This typically creates holes in shadow maps. The holes are filled in another stage with the use of pull-push algorithm [MKC07]. The figure 4.10 shows a scene with two virtual point lights and their ISMs.

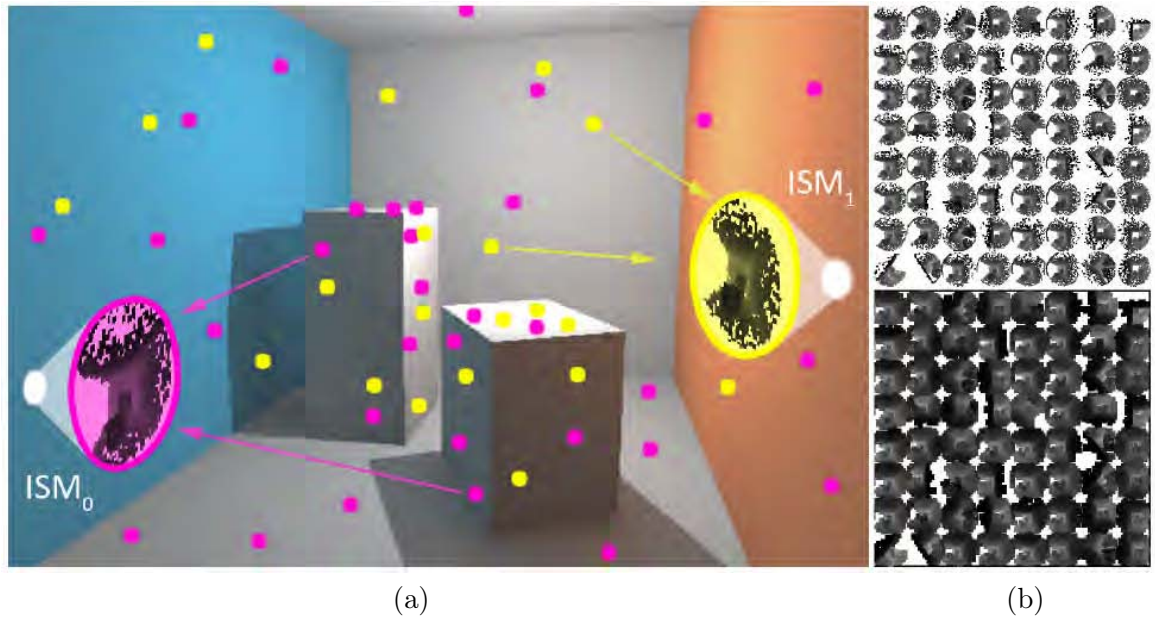


Figure 4.10: A scene with two virtual lights and associated ISMs (a); the top and bottom ISM textures (b) show shadow maps without and with pull-push phase respectively. Courtesy of Ritschel et al. [RGK⁺08]

During the pull phase the image pyramid is built from the ISM texture by scaling the texture down by the factor of two. But only valid depths are combined (averaged) when computing the coarser level. In the push phase holes are filled with interpolated values from the coarser level. ISM use outlier rejection during both pull and push phases.

As ISMs are rendered in one pass, they are stored in one big texture. This texture typically has the resolution 4096×4096 pixels. If the resolution of one ISM is 128×128 pixels, the big texture can contain 1024 Imperfect Shadow Maps. To cover full hemisphere, the ISM algorithm uses parabolic mapping.

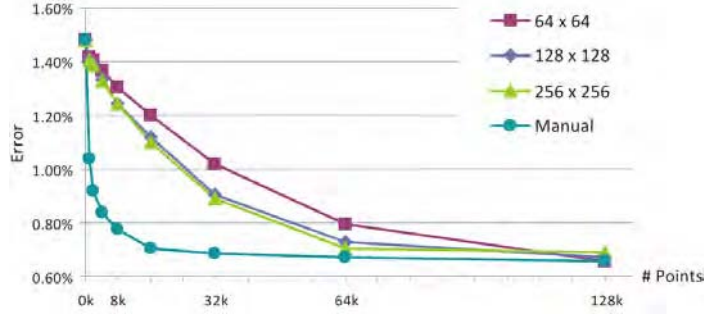


Figure 4.11: Normalized RMS image error of the Sponza scene; the ISM resolution is varied as well as the number of point samples; courtesy of Ritschel et al. [RGK⁺08]

If the ISMs are used to approximate the visibility in the scene, one major problem may occur. As the original ISM implementation tries to distribute point samples uniformly over the scene, for a huge and complex scene the number of point samples can become impractical. On the other hand, if the number of point samples is kept low, visibility artifacts may occur. The View-Adaptive Imperfect Shadow Maps (VAISM) algorithm is designed to remove this major limitation of ISM [REH⁺11]. Another challenge for this algorithm is to sample VPLs more wisely.

As Bidirectional Instant Radioisity [SIMP06], this algorithm prioritizes VPLs which contribute more to the final image. The goal is to create VPLs only where they can influence pixels of the final image. The algorithm uses reflective shadow maps, so each pixel of RSM can potentially be a VPL. The optimal algorithm would test visibility from all pixels of RSM to whole surface seen from the camera. However, this solution is impractical due to high computational requirements.

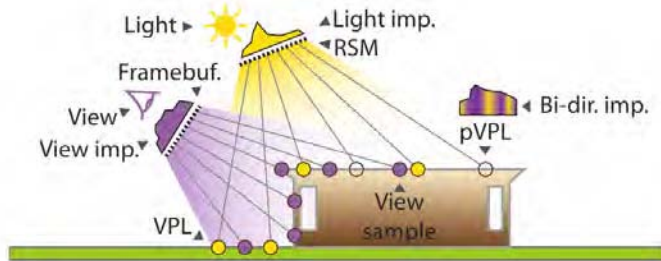


Figure 4.12: Bidirectional Reflective Shadow Maps; courtesy of Ritschel et al. [REH⁺11]

To overcome this limitation the VAISM algorithm uses two reflective shadow maps, one rendered from the camera, the second rendered from the light position and evaluate the influence of VPLs stochastically. For each potential VPL only a random subset of samples is taken into account. Visibility query is omitted to speed the evaluation up. Only mutual position and orientation is used to estimate the influence.

To reduce the number of needed point samples over the scene, this algorithm breaks the uniformity of the point sample distribution. In ideal case, only triangles blocking the light

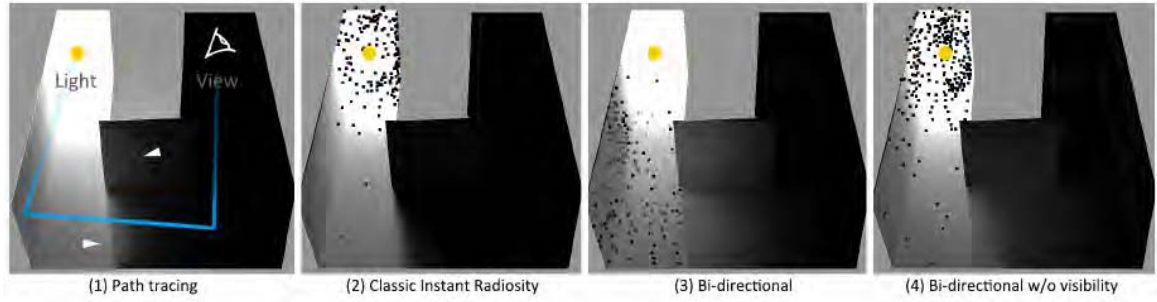


Figure 4.13: Comparison of various VPL distributions; the classic instant radiosity (2) places samples around the light source. The Bidirectional Instant Radiosity produces much better VPL distribution. The VAISM algorithm produces worse VPL distribution than BDIR, but can be evaluated more effectively. Courtesy of Ritschel et al. [REH⁺11]

should be given some point samples. Distant triangles, which do not contribute to the ISM texture, do not require any point samples. The problem is to estimate which triangle does influence the viewable light transport in the scene. The VAISM algorithm estimates the influence by computing the solid angle between the visible scene point and the triangle.

Correct solution would compute the solid angle between all triangles and all visible scene points, but this is also impractical. The algorithm uses stochastic sampling and selects several visible points for each scene triangle. The triangles with bigger solid angles receive more point samples from those with lesser solid angles. To avoid flickering only a subset of scene points is updated every frame. This leads to temporal inaccuracy, but it seems not to be disturbing for human observers [REH⁺11].

Chapter 5

Other Interactive Global Illumination Algorithms

In this chapter we mainly discuss algorithms that are not based on the Instant Radiosity approach. However, they share many characteristic properties with the Instant Radiosity algorithms. The biggest similarity is the targeted segment of graphical applications. All algorithms reviewed in this chapter are optimized for fast rendering, the physical correctness is inferior.

In the section 5.1 we review the Cascaded Light Propagation Volumes algorithm. The rest of this chapter investigates the screen space illumination approximations.

5.1 Cascaded Light Propagation Volumes

Cascaded Light Propagation Volumes (LPV) were developed by Kaplanyan et al. in 2010 [KD10]. They are targeting interactive applications, so the algorithm is not physically correct. LPV works best on low-frequency indirect shading. The direct illumination is computed separately with the help of classic shadow maps.

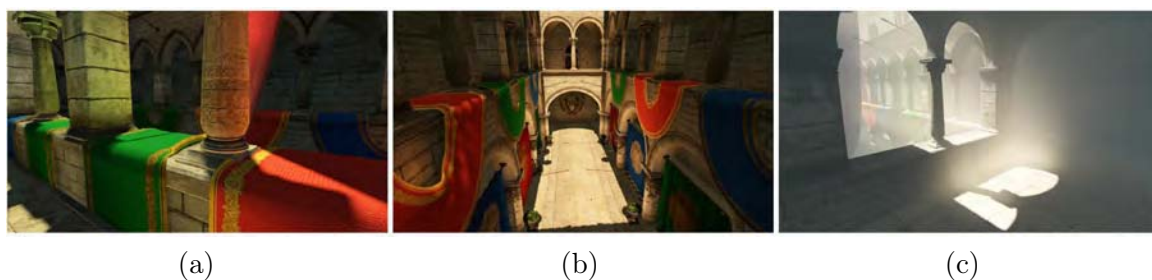


Figure 5.1: The "Crytek Sponza" scene rendered using LPV (a) (b); the image (c) shows a simple participating media scattering. Courtesy of Kaplanyan et al. [KD10]

The basic idea is similar to grid based fluid advection simulations. The algorithm creates two grids, one for geometry approximation and one for indirect light intensity. Both grids are filled every frame from scratch. The geometry grid is initialized from the camera geometry

buffer (g-buffer) and from the light reflective shadow maps. To speed this step up, static geometry samples can be reused from the last frame. The light intensity grid is filled with data gained from reflective shadow maps and from low frequency light sources such as area lights or environment maps. In each cell the intensity is discretized using low-order spherical harmonics.

The algorithm is iterative and in each iteration the flux coming from each cell to its neighbours is computed. The algorithm uses just the base six directions aligned to the axes to exchange the flux. The geometry grid is used to compute the blocking factor between adjacent cells. After heuristically determined number of iteration is reached the intensities are used to illuminate the scene.

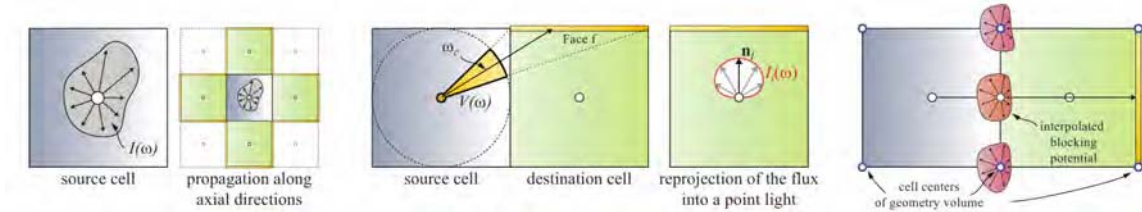


Figure 5.2: The propagation scheme and the flux estimation for adjacent cells in LPV; courtesy of Kaplanyan et al. [KD10]

The algorithm creates nested grids to catch more detail in parts closer to the viewer. As the algorithm recreates the grid every frame, the nested parts are dynamically adapted as the camera and scene objects move. The algorithm can easily be extended to capture some participating media effects, as shown in the figure 5.1.

5.2 Screen-Space Illumination Approximations

Fast global illumination estimation over the scene is still challenging even with the current hardware. One of the popular approximations of global illumination effects is ambient occlusion (AO). This method basically just darkens holes and corners because it assumes lower indirect illumination coming to such places. In static scenes AO darkening term can be precomputed and stored as per vertex attribute or in texture. But if we deal with dynamic geometry, the precomputed AO can easily become inaccurate.

One example of real-time AO for dynamic scenes was presented by Kontkanen and Aila in 2006 [KA06]. They present new AO algorithm for animated models, typically characters. The AO is precomputed for several key poses of the model. Later the algorithm interpolates the precomputed references and creates AO for the animated model in real-time.

5.2.1 Screen-Space Ambient Occlusion

The full-scene ambient occlusion evaluation typically involves ray tracing which currently does not scale well to the full dynamic scenes used in applications like computer games. But recent research shows that screen-space approximation of AO can produce pleasant results in real-time frame rates [RGS09] [BS09].

The principle of SSAO is shown in the figure 5.3. The depth map samples (green and blue) are used as a heightfield. The AO is approximated by ray-marching several rays from the actual point P in the direction randomly chosen on hemisphere given by normal of the surface point P . If the depth stored in the buffer is greater, the ray continues. But in the other case (the rightmost sample in the figure 5.3) the ray intersects the surface and the occlusion is accumulated. The number of samples the same as well as the length of rays can be varied to meet performance and quality requirements. The noise generated by random samples is often filtered out using geometry aware blur [SIP06].

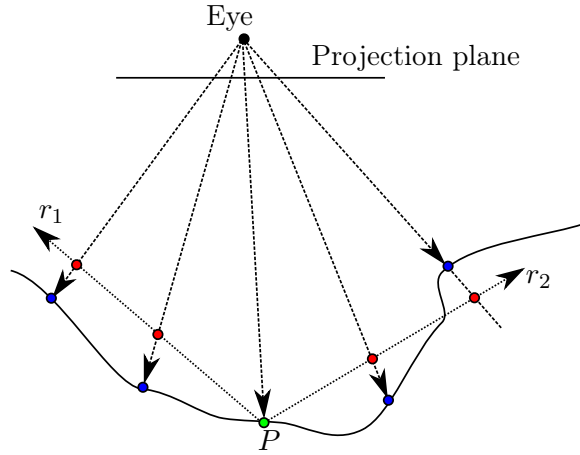


Figure 5.3: Computation of SSAO for the point P (green) with two sample rays r_1 and r_2 .

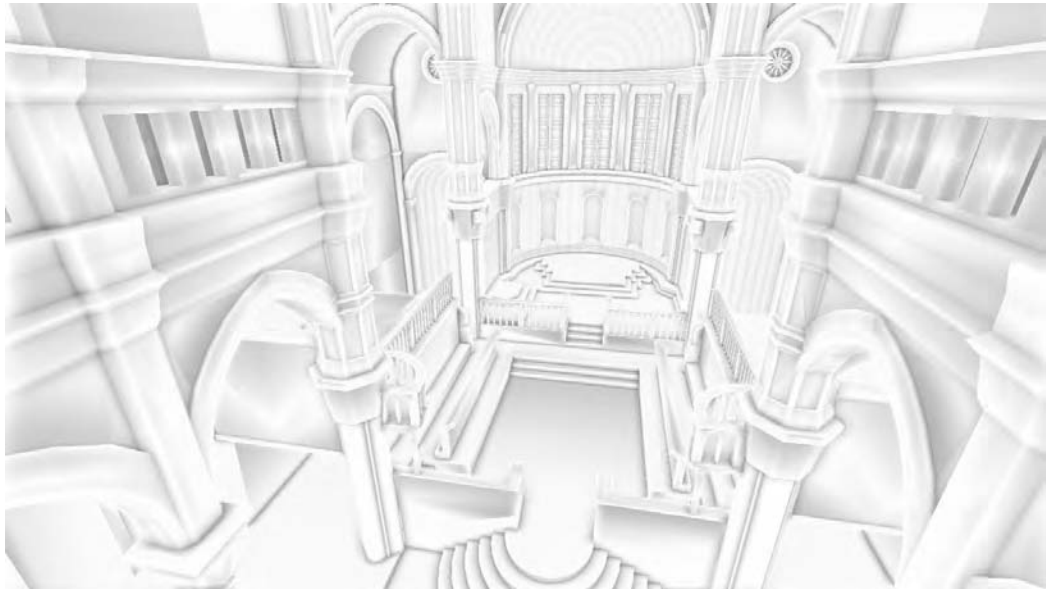


Figure 5.4: Ambient occlusion in the Sibenik Cathedral scene computed on-line using the screen space approach; the image was rendered in the resolution of 1280×720 pixels in 1.39 ms.

The SSAO algorithm suffers from several shortcomings. The biggest problem occurs when the depth map does not contain needed surface information. This scenario occurs for example when the occluder samples are overdrawn by much nearer object. In this case AO cannot be correctly evaluated. There are several workarounds used to overcome this problem. Multiple depth layers can be created and sampled [BS09]. Or multiple views can be rendered [RGS09].

The SSAO gives us several great features. As it is computed in image space, it does not depend on the scene complexity. SSAO only requires normal and depth images, so it can be used with almost any kind of the rendering algorithm. It also works very well with normal mapping and all displacements techniques. The implementation of SSAO is straightforward and it only has to be added as a post-processing stage or a deferred stage.

5.2.2 Screen-Space Directional Occlusion

The standard AO techniques usually separate the AO step from the other illumination steps (like direct illumination). So the AO map usually contains only gray-scale values and does not distinguish various illumination sources which were occluded. The final image is often created as a product of AO gray-scale map with the illuminated scene (without AO). The algorithm tries to merge those steps together. During the evaluation of AO for the point P if any occluder is found, the radiance incoming from the same direction as the occluder is blocked. The Directional Occlusion can be used with many types of known illumination sources; like environment map or point lights.

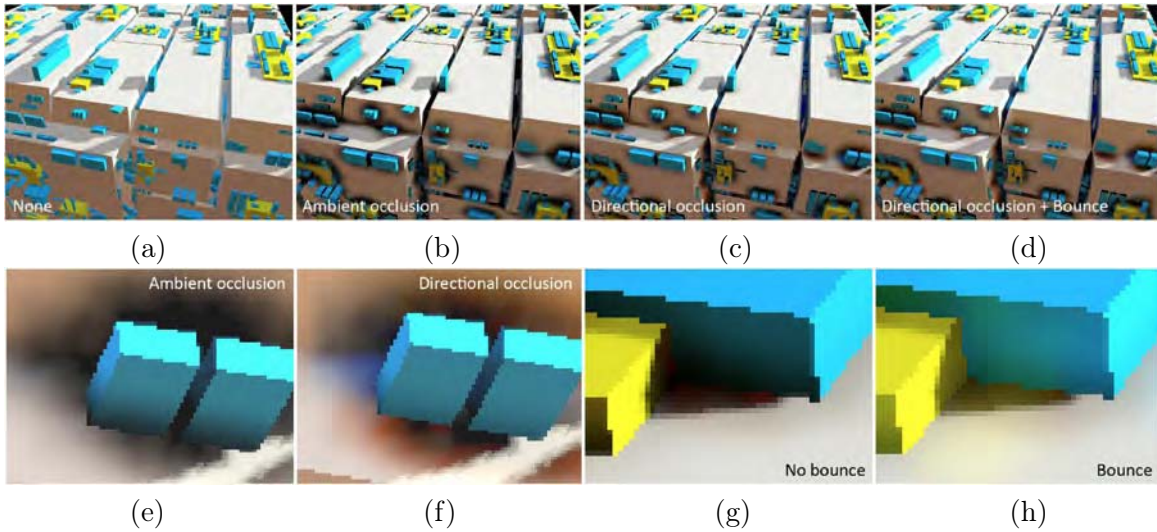


Figure 5.5: Comparison of image without AO (a), standard SSAO (b), SSDO (c) and SSDO with a diffuse bounce (d); the second row shows the details of SSAO (e), SSDO (f) and (g), and SSDO (h). Courtesy of Ritschel et al. [RGS09]

Occluders detected in the AO step can also be used to cast radiance to the point P . This approximation of one light bounce produces really pleasant images at the cost of small computational time [RGS09]. The results of SSDO and screen space indirect bounce can be seen in the figure 5.5.

Chapter 6

Problem Analysis

In this chapter we take a deeper look at the possibilities of state of the art global illumination algorithms for dynamic scenes. We have reviewed all major known algorithms in the chapter 4. In this chapter we investigate the demands of our assignment, analyse so far known instant radiosity algorithms, and we give our reasons for chosen technologies and programming languages. We discuss the design of the application itself in the following chapter 8.

6.1 Analysis of the Assignment

As the research shows, the indirect illumination evaluation is usually vital if we want to produce perceptually pleasant images [Sto04]. Also it is known, that the diffuse reflections play the most important role in the scene illumination in the most of the scenes [Sto04]. If we want to produce interactive applications, we have to carefully choose which components of the illumination we will take into account. The complete evaluation of physically correct lighting model is still unconquerable in the interactive applications.

Radiosity algorithms were designed to deal mainly with the diffuse surfaces. From the "quality versus speed" point of view, omission (or coarse approximation) of the specular and glossy indirect lighting can be a good choice. In addition, instant radiosity can take advantage of the hardware accelerated rasterization. GPU friendly algorithms seem to be the right choice in general, since there is a significant boost in GPGPU technologies in recent years.

The task is to use one of the instant radiosity algorithms to calculate the global illumination. Many of the instant radiosity algorithms have the potential for rendering global illumination in interactive frame rates. On the other hand, physical correctness often suffer when the algorithm has to be fast.

Another big challenge is the rendering of dynamic scenes. When the scene is fully dynamic (objects, cameras and light sources move frequently), little or no pre-computation can be used. This fact limits many ray-tracing based algorithms as they usually require hierarchical acceleration data structures which are difficult or impossible to update. On the other hand, rasterization seems to work very well with dynamic scenes. Albeit it is a common practice

to use acceleration data structures with rasterization too, however they often cover the scene only in a coarse manner. So no fine refinements are usually needed every frame.

Last but not least requirement is the platform independence of the implementation. We require the measurement to be done on various setups thus the code has to be portable.

Our summarized requirements are:

1. the algorithm has to handle fully dynamic scenes,
2. one frame should be rendered within the order of tens of milliseconds,
3. one indirect diffuse bounce is sufficient¹,
4. the algorithm should be well parallelized and portable to GPU,
5. the implementation has to be written using only portable libraries.

6.2 Algorithm Selection

Tremendous amount of work has been done in the field of the global illumination. If we deal with a static scene, we can easily involve preprocessing. A typical solution pre-computes the visibility or radiance transfer. An example of the visibility preprocessing can be found in the Coherent Shadow Maps algorithm [RGKM07] and the successive Coherent Surface Shadow Maps method [RGKS08]. The radiance transfer preprocessing is used in the algorithm presented by Szécsi et al. [SSKS06].

As the visibility or radiance transfer cannot be pre-computed for all pairs of the scene points, the interpolation and filtering is usually used. If problems caused by scene discretization in the preprocessing stage are solved, these methods produce highly pleasant images that are close to the physical reality. However, the use of pre-computation is usually heavily limited in dynamic scenes.

Instant Radiosity elegantly solves this issue by using the Monte Carlo estimation of the indirect illumination [Kel97]. The algorithm successfully approximates lighting in diffuse scenes with many virtual point lights. But the original algorithm still requires a huge amount of computational power to evaluate the visibility of created VPLs. One of the possible improvements exploits the temporal coherence between successive frames. For example, Incremental Instant Radiosity presented by Laine et al. in 2007 [LSK⁺07] reuses virtual point lights from previous frames and it excludes dynamic objects from the visibility steps. Thus, all scene objects (including dynamic) receive the indirect illumination, but only static objects can contribute to it.

Another approach simplifies the visibility queries for the indirect paths. Imperfect Shadow Maps, as presented by Ritschel in 2008 [RGK⁺08], substitute the scene with an approximate representation to speedup the visibility step. ISM are improved to handle large complex scenes by Ritschel in 2011 [REH⁺11]. ISM produce medium quality results with a budget of only a few milliseconds on a modern hardware.

¹One indirect bounce means two bounces total.

We have chosen the ISM algorithm for our implementation. It creates a good trade-off between the quality and the rendering time. It also has no problems with dynamic scenes. As we show in the following text, the point representation can be rebuild from scratch in every frame using the hardware accelerated tessellation. We would like to explore the potential of combining the ISM based algorithm with a screen space global illumination approximation, as ISM cannot handle small local light events. Ritschel et al. gained highly pleasant results using their Screen-Space Directional Occlusion [RGS09], presented in 2009. Screen-space methods gained big popularity in recent years, as they require no preprocessing and successfully overcome the scene complexity.

6.3 Framework Selection

In this section we will describe the application needs one by one, with the discussion about the individual library/language candidates. The first choice had to be made about the programming language for the basic application. The decision was really simple here. As we require a lot of specialised libraries to be usable with our application, we have chosen C/C++ language. The majority, if not all, of the libraries we use, is written in C/C++ language, so we can use native bindings without the need of any wrappers. C/C++ language also gives us adequate comfort and performance.

The biggest decision was made around the graphics programming API to use. Since the algorithm is almost a perfect example of GPU friendly system, it would be a nonsense to overlook the GPU acceleration possibilities. First we had to choose between a general purpose rasterization API and a specialized hand-coded library in a GPGPU language. Also we wanted to minimize the total number of used languages and keep the code maintainable. From the systematic point of view, one common language, for the rasterization and the computations, would be ideal. However, recent research shows that hand-coded software rasterization is at average approximately five times slower than a hardware accelerated engine [LK11], even when written in CUDA to run on GPU.

The performance difference between the CUDA software rasterization engine written by Laine et al. and common hardware accelerated libraries convinced us to use a general purpose hardware accelerated API for rasterization. The biggest players in the field of accelerated rasterization are DirectX and OpenGL libraries. DirectX comes with a bundled general purpose language called DirectCompute. The relationship between DirectX and DirectCompute is really close, since DirectCompute was primary designed to cooperate with DirectX. This is surely a big advantage as the approach and design patterns are similar.

However, DirectX is strongly bound to Microsoft platforms. This does not fulfill our portability requirement. The second mentioned API - OpenGL, has no such restrictions, in fact it is adopted to almost all modern architectures and operating systems [Khr12c]. The functionality provided by the OpenGL API covers our needs at ease. Also OpenGL shading language (GLSL) can be used to run our computations. So we have chosen OpenGL as our main graphic framework.

OpenGL does not come with two important things: resource loading subsystem and the window and input event handling functionality. There are many OpenGL compatible libraries created to simplify the window and input handling. For example GLUT library [Khr12a], which is tightly connected to OpenGL, offers this functionality. However, it

is a really simple library with very limited GUI creation support. There was a possibility to use more libraries for the GUI and window management. For example GLUT for window management and FLTK [Bil12] library for the GUI creation. But it would add more complexity to the whole project, so we decided to use only one complex framework which would cover GUI creation and window management altogether.

There are several multi-platform GUI libraries that support OpenGL. We have considered three candidates: GTK+ [GTK12], Qt [Nok11] and wxWidgets [wxW12]. All the candidates are well-known mature GUI frameworks for C++ and other languages. We have chosen Qt libraries as they provide really great documentation, stable and clean API, and good performance and memory footprint.

For the scene loading, we have found the Open Asset Import Library [Ass09]. The huge benefit from using the Assimp library is the list of supported formats. The Assimp library supports dozens of scene formats, such as `obj`, `ply`, `dae`, `3ds`, `x` and others. The library also supports parsing material properties, for example the `mat` files.

For the image loading and storing, we have chosen the DevIL library [Dev12]. The other adepts were the CImg [Tsc12] library and the FreeImage [Fre12] library. DevIL has a really simple OpenGL-like API and supports a wide range of image formats, such as `png`, `jpeg`, `exr`, `hdr` and many others [Dev12].

The last third party software, we use, is the GLEW library [GLE12]. We use GLEW to simplify the OpenGL extension and version handling.

Chapter 7

Used Technologies and Libraries

Our application is built on several programming languages and supporting libraries. As the majority of computations runs on the graphics processing unit, consequent sections focus on GPU computing technologies such as CUDA or GLSL. We briefly introduce NVIDIA CUDA platform, we also describe used graphics library and its shading language.

7.1 GPU Computing and CUDA

The demand for high quality realistic game rendering has pushed GPU vendors to create high performance computing architectures. The computer graphics tasks were always characterized as highly parallel, data and arithmetic intensive applications. NVIDIA CUDA is a high level C++ like programming language and framework for writing general purpose applications that run on the GPU. CUDA has become very popular as it brings the horsepower of modern GPUs to all kinds of highly parallel computational intensive applications, such as ray tracing [ZH10].

7.1.1 Today's GPU architecture

As can be seen in the figure 7.1, the GPU architecture strongly differs from the CPU architecture. The CPU typically contains only a few arithmetic units which are surrounded by a lot of control logic and caches. This perfectly meets typical desktop application requirements as desktop applications usually work only with one or a few threads. Also, desktop



Figure 7.1: Comparison of typical CPU architecture (left) to the GPU architecture (right); courtesy of nVidia [NVI10]

applications often perform a mix of logic and arithmetic operations and jump "randomly" over the memory which requires a lot of cache. The thread scheduling is generally done by the operating system and that creates another complexities in the whole architecture.

On the other hand, graphical tasks are usually highly parallel (there are for example millions of pixels to evaluate but each pixel can be evaluated independently of the other pixels). Also, at some point all the threads typically execute the same code (for example all threads are transforming a vertex to the camera coordinate system). The memory access pattern differs from the CPU applications too. For example, the image filter kernel typically reads surrounding pixels, and if two kernel instances are executed on neighbouring pixels, there is a high chance that they will read and write similar locations in the memory. The thread scheduling can be often really simplified, as there is no complicated priority mechanism, much lower number of active tasks or no complex memory protection required.

In the figure 7.2, we can see a block diagram of one Streaming Multiprocessor (SM) used in NVIDIA Fermi architecture [NVI09]. One SM usually contains tens of arithmetic logic units, several special function units and tens of load/store units. The register file is used as local per thread memory. Shared memory can be used by all threads running on the same SM, but is not visible to threads scheduled to run on the other SMs. The thread scheduling is done by The Warp Scheduler unit. As the scheduling is implemented in hardware, it can switch threads really fast [NVI09]. Streaming Multiprocessor usually contain only a limited size cache.

The small cache cannot be used alone to hide the latency of main memory, which is usually hundreds of GPU cycles [NVI09]. To overcome this limitation, programmers should use the shared memory and try to avoid random accesses to the main memory. Also The Warp Scheduler detects memory accesses and reschedules threads waiting for the memory operation to complete. This, however, works only if sufficient number of threads is available to the scheduler.

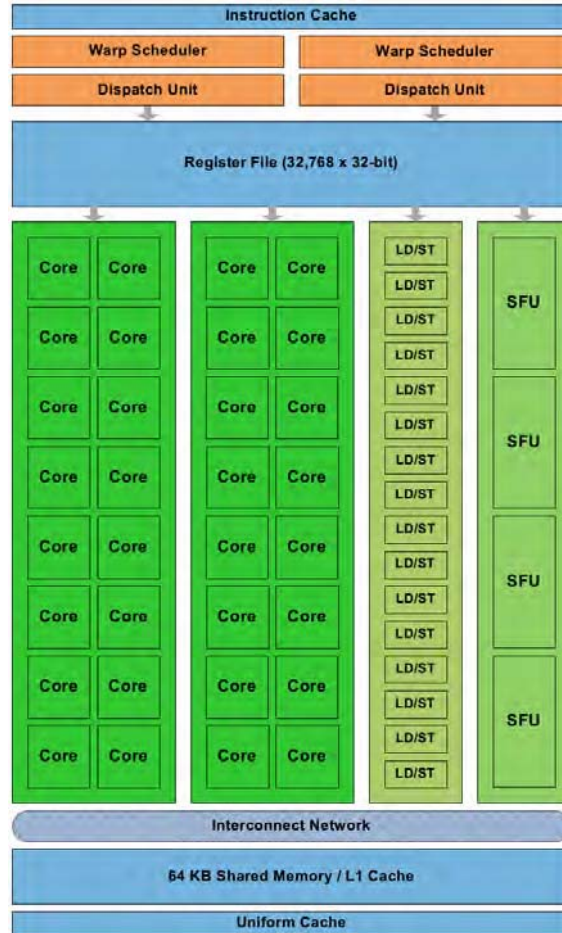


Figure 7.2: One Fermi Streaming Multiprocessor; courtesy of nVidia [NVI09]

7.1.2 Sample Code

As mentioned before, CUDA programmers have to be aware of the architectural principles and the memory layout to be able to write efficient GPU code. But the language itself offers a high level approach to simplify kernel writing as much as possible. The kernel code (the code which runs on GPU in many instances in parallel) is written directly into the host code. This feature allows developers to write structures, classes and methods which are shared across the GPU and CPU environment. CUDA compiler also extends standard C++ host code to simplify kernel invocations and serial to parallel code transitions.

In the figure 7.3 we show a simple CUDA application skeleton. The application performs a parallel addition of arrays A and B and stores the result into the array C. The `__global__` keyword denotes kernel function and the `<<<>>>` operator indicates kernel invocation from the host code. Kernels can read built-in variables, such as `threadIdx`. The rest of the code is written against the C++ standard.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
// Application entry point
int main()
{
    // Memory initialization
    initializeArrays(A, B, C);
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    // resource deallocation, ...
}
```

Figure 7.3: Simple parallel vector addition; an example from [NVI10]

7.2 OpenGL and GLSL

OpenGL is a well known standard for hardware accelerated 2D and 3D rasterization library. It was introduced in 1992 and has been adopted to all major platforms such as Mac OS, Linux or MS Windows [Khr12c]. The API is written in the C programming language, but it is extended to all major languages such as Java, Python and Perl. OpenGL is partially object oriented, as there are functions altering the global state mixed with functions that work purely with associated objects¹.

¹As C syntax does not support classes and methods directly, everything in OpenGL is a global function or constant.

Recent versions of OpenGL are tightly connected to their shading language, named OpenGL Shading Language (GLSL). GLSL is a C-like language, designed to perform graphical computations on parallel architectures such as today's GPU. GLSL programmers work on higher level of hardware abstraction than for example CUDA or OpenCL programmers do, since the language is much more hardware independent. The language was primarily developed to cooperate with the rest of OpenGL, so the interoperability between OpenGL and GLSL generally works better than between OpenGL and CUDA which we discuss in the section 7.3.

OpenGL was also used as a base of OpenGL ES, which is a modern graphics API for embedded devices [Khr12b]. Last but not least adoption of OpenGL is the WebGL API which brings accelerated graphics interface to the web browsers through the ECMA script [Khr12e].

7.2.1 History of OpenGL and GLSL

From the beginning OpenGL was designed to keep the backward compatibility, as it was needed by many industrial applications. However, after almost two decades of intensive evolution of graphics hardware, OpenGL has become too complex and hard to implement, because it had to support all the historical features next to the modern approaches [Ope11]. To solve this issue, OpenGL 3.0 introduced a deprecation model and OpenGL profiles.

In the table 7.1 we show all OpenGL versions released until May 2012. Since OpenGL 2.0 release in 2004, the GLSL language has become an important part of OpenGL². GLSL 1.10 created two programmable stages in the OpenGL pipeline. In all versions before the 3.1 release, shader programs were only optional fixed functionality replacement. Shaders were

Version	Year	Notable features
1.0	1992	first release
1.1	1992	vertex arrays
1.2	1998	3D textures
1.2.1	1998	ARB extensions
1.3	2001	cube maps, multitexturing
1.4	2002	depth textures
1.5	2003	buffer objects, occlusion queries
2.0	2004	GLSL 1.10, point sprites, MRT
2.1	2006	GLSL 1.20, PBO ³
3.0	2008	GLSL 1.30, deprecation, profiles, transform feedback ⁴
3.1	2009	GLSL 1.40, instanced rendering
3.2	2009	GLSL 1.50, geometry shaders
3.3	2010	GLSL 3.30, sampler objs., timers
4.0	2010	GLSL 4.00, tessellation, per-sample sh.
4.1	2010	GLSL 4.10, shader binaries, 64bit sh.
4.2	2011	GLSL 4.20, atomics on buffers

Table 7.1: Official OpenGL releases with some of the features they introduced

²GLSL could be used in older versions too, but only using the extension.

usually connected to many built-in variables and interfaced frequently with the rest of fixed function pipeline.

7.2.2 Modern OpenGL Pipeline

In the OpenGL 3.2 release, the fixed functionality was minimized and the old pipeline was dropped [Ope11]. That introduced great flexibility to GLSL and allowed writing a general purpose code in shaders. In later versions of OpenGL, another shading stages were added to meet demands of 3D graphics programmers.

The actual shader pipeline is shown in the figure 7.4. The vertex data is read by a vertex shader which typically calculates vertex transformations. In the simplest case, transformed vertices are assembled into primitives, calculated attributes are interpolated and sent directly to the rasterization process. If the tessellation evaluation shader is present, primitives can be further tessellated (or discarded) depending on tessellation control shader. Geometry shader can be used to calculate per-primitive properties or to change primitive types (and count) just before the rasterization process. The final stage involves fragment shaders which evaluate properties of the future pixels.

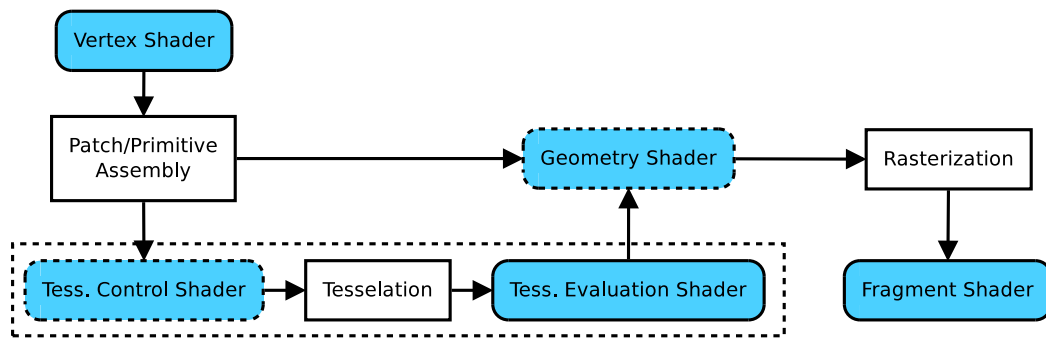


Figure 7.4: Simplified OpenGL pipeline; the programmable stages are shown as light blue rectangles. Dashed rectangles denote optional stages.

7.2.3 OpenGL Shading Language

GLSL is a C-like high level programming language, designed mainly to perform graphical computations with tight connection to OpenGL. The language itself has the same syntax across all shading stages. The difference between the particular stages is only in shader inputs and outputs and in the way which shaders are executed.

Unlike CUDA, GLSL does not provide low level data access mechanisms, such as pointers [Khr12d]. The data mapping is partially hidden from the programmer by the GLSL implementation. However, reading any kind of memory was always possible using textures and samplers. In recent GLSL versions all shader stages can read and write any part of allocated memory buffers using image objects. Also atomic operations and synchronization primitives were added to GLSL.

In contrast to CUDA, GLSL programmers have almost no possibility to influence the organization of GLSL program launches. The number of threads per block/multiprocessor

is determined automatically by the OpenGL implementation. However, the architectural characteristics influence the GLSL environment as well. For example, the amount of shared memory available to CUDA kernels is typically the same as the maximum size of GLSL uniform buffers.

7.2.4 GPU Accelerated Tessellation

Functionality added to OpenGL 4.0 allows programmers to write tessellation shaders. Tessellation is used to subdivide incoming primitives and create finer level of detail on the fly. This technology was designed to spare memory bandwidth and space, as there is no need to store (and transfer) many levels of detail of the geometry. Also shaders can finely tune tessellation levels instead of switching several precomputed geometries for each level of detail.

Tessellation shaders interact with built-in tessellation unit which accelerates the primitive subdivision. The tessellation level can be evaluated by the tessellation control shader⁵ (TC), which is optional. If the TC shader is not present, tessellation level has to be set by the CPU code and remains the same for the whole draw call. The built-in tessellation unit then subdivides input primitives, the example tessellation of a triangle is shown in the figure 7.5. TC shader can be also used for per primitive culling. Using zero (or less than zero) tessellation level value discards the primitive.

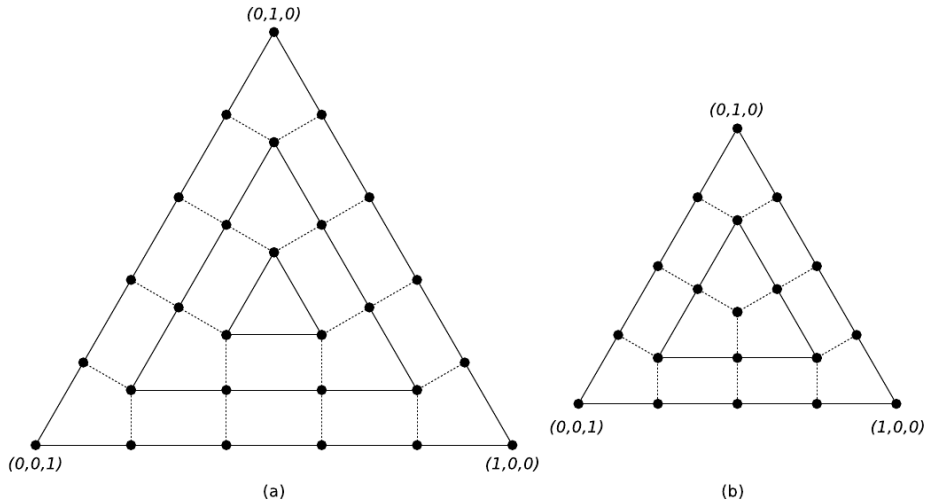


Figure 7.5: Triangle tessellation as done by OpenGL; the triangle (a) has the tessellation level of five, the triangle (b) has the level of four. Barycentric coordinates are written next to their corresponding vertices. Courtesy of The Khronos Group [SA11]

Tessellation evaluation shaders (TE) receive interpolated tessellation coordinates (barycentric coordinates in the case of triangle tessellation) and execute over newly created primitives. TE shaders typically evaluate properties of new primitives, the common practise is to use displacement mapping with online tessellation together.

⁵See the figure 7.4 for the details of shading pipeline.

7.2.5 Sample Code

The first code snippet shows the body of a simple OpenGL drawing loop. The frame buffer is cleared, shader program is selected, vertex data settings are applied and triangles are drawn. The last line of code sample contains a platform dependent code. To hide the platform dependent functionality, as frame buffer switches are, OpenGL library is often combined with another supporting libraries, such as GLUT.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glUseProgram(shaderId);
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, nTriangles * 3, GL_UNSIGNED_INT, 0);

glXSwapBuffers(...); // platform dependent
```

Figure 7.6: A simple OpenGL drawing loop

The following lines of GLSL code in the figure 7.7 were taken from our implementation (from the geometry stage of deferred rendering). The simple vertex shader translates and projects the input position on the screen. Normal vector it transformed by the normal matrix, texture coordinates are just written through.

```
#version 330
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 nor;
layout(location = 2) in vec2 tex;

uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

smooth out vec3 nor_v;
smooth out vec2 tex_v;

void main()
{
    nor_v = normalMatrix * nor;
    tex_v = tex;
    gl_Position = modelViewProjectionMatrix * vec4(pos, 1.0f);
}
```

Figure 7.7: A simple GLSL vertex shader

7.3 CUDA and OpenGL Interoperability

If an application uses more GPU programming frameworks together, such as OpenGL/GLSL and CUDA, mechanism for efficient sharing of resources is needed. In the worst scenario, developers are forced to copy data to the host RAM and then write it back to the GPU. For example, an OpenGL rendered image is downloaded and then uploaded as a CUDA array. This approach causes stalls in the GPU as well as in the CPU execution, and it introduces unnecessary data moves.

CUDA tries to solve this by introducing the mechanism of shared resources. The CUDA context has to be initialized with a connection to an existing OpenGL context, then memory buffers and textures can be shared across the CUDA and OpenGL environments. However, we observed that not all important OpenGL image formats are supported in CUDA. This issue is discussed in the section 8 in more detail.

7.4 Another Supporting Libraries

OpenGL itself does not provide any functionality for window and input management, it focuses purely on graphic operations. Also graphic resources, such as models and textures, are not loaded directly by OpenGL. To keep our application portable, we have decided to use only platform independent libraries. For window and input management, we use the Qt libraries. Image loading is done by the Devil library (sometimes also called OpenIL). To simplify model and scene loading, we have used Open Asset Import Library. The OpenGL extensions are managed by the GLEW library.

7.4.1 Qt Libraries

Qt libraries create an advanced application creation framework for GUI and console programs. The framework itself is written in C++, but bindings to another languages exist. Qt is popular mainly for its clean and consistent API and great documentation [Nok11]. The most known parts of the Qt framework are the QtGUI, QtCore, QtOpenGL and QtXML libraries. The framework is ported to many desktop and mobile architectures.

A simple Qt application is shown in the figure 7.8. Qt framework is characteristic with its signal/slot event propagation mechanism. As C++ does not provide the needed functionality,

```
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton btn("Hello World!");
    btn.show();
    return a.exec();
}
```

Figure 7.8: Simple Qt application which creates a button with the "Hello World!" sign.

Qt authors created, as it is called, Meta Object Compiler (MOC) which pre-parses the source code and generates a helper C++ code, which is then compiled together with the rest of the application. The MOC also adds a higher language features to C++, such as automatic memory management or property system.

7.4.2 DevIL

DevIL is a simple yet powerful image loading library. It is written purely in the C programming language and supports a wide range of image formats [Dev12]. The library should be easy to learn for OpenGL developers since it uses similar naming conventions. DevIL is shipped with an utility library, which adds a basic image modification functionality, such as scaling.

```
ILuint img_id;
ilGenImages(1, &img_id);
ilBindImage(img_id);
if(ilLoadImage("lenna.jpg") != IL_FALSE)
{
    ILint w = ilGetInteger(IL_IMAGE_WIDTH);
    ILint h = ilGetInteger(IL_IMAGE_HEIGHT);
    // ...
}
```

Figure 7.9: Sample code that uses DevIL library to load an image from a file; if the operation succeeds, the width and height of the image are retrieved.

7.4.3 Open Asset Import Library

The loading of models and scene descriptions in our implementation is done by The Open Asset Import Library (Assimp). The library is capable of loading numerous scene formats [Ass09]. Also all the materials, animations and a scene graph structure is parsed and accessible after a successful import. Assimp supports various post-processing steps, such as triangulation, normal generation or mesh optimization.

```
Assimp::Importer imp;
const aiScene * scn = imp.ReadFile("bunny.obj", aiProcess_Triangulate);
if(scn) {
    for(unsigned m = 0; m < scn->mNumMeshes; ++m) {
        // process mesh
    }
}
```

Figure 7.10: Loading a scene with Assimp

7.4.4 GLEW

To simplify the OpenGL extension handling, we use The OpenGL Extension Wrangler Library (GLEW). GLEW effectively hides all the tedious code which is needed to check and load the OpenGL extensions. The usage of GLEW is really simple. After the library is initialized, developers can check the availability of certain OpenGL versions simply by testing the corresponding variable.

```
if(glewInit() != GLEW_OK) {
    std::cerr << "Cannot initialize GLEW!" << std::endl;
    exit(1);
}
if(!GLEW_VERSION_3_3) {
    std::cerr << "OpenGL 3.3 or higher not available!" << std::endl;
    exit(1);
}
// continue with OpenGL 3.3 and higher
```

Figure 7.11: Code snippet showing GLEW usage

Chapter 8

Implementation

In this chapter we describe our implementation in detail. The application is written in the ANSI C/C++ programming language and we employ Qt libraries for the window, GUI and OpenGL management. Non-negligible part of the code is written in CUDA and GLSL and runs on a GPU. The CPU code is written only against cross-platform libraries (Qt, OpenGL, GLEW, etc.), but as we use CUDA, the application is bound to platforms with nVidia graphics cards with compute capability 2.1 [NVI10].

The CPU code does minority of the computations during the rendering. So we have focused on the GPU part of the code. That means, the CPU code does not contain any deep optimizations. We do not use any vector arithmetic instructions, such as SSE. We use a simple path-tracer as a reference solution. This module is also mostly written in the CUDA language.

We use CMake [CMa12] as our build system. So users are free to generate project files (or Makefiles) for their favourite development environment. Installation is described in the section C. User manual and build instructions are included as well.

As the rendering pipeline is based on the ISM algorithm, prior knowledge of the algorithm is important. The ISM algorithm is discussed in the section 4.11.

8.1 Application Structure

Application is written using the C++ language. The computer graphics application designs are often object oriented, as the problem itself suggests. We have the Light class, Camera class, etc. Our implementation follows this common practice and is mainly object oriented. However, too deep levels of structuring and abstraction can introduce performance drops. So we have designed our application not only with respect to the code readability, but also with respect to the performance.

Our goal was also to follow the mode-view-controller pattern. This adds a little of complexity to the whole design, but as a result, the resource reading and scene management is completely independent of scene rendering. This creates the possibility of writing new render back-ends without altering the scene reading or managing code. So our ISM renderer and path-tracing renderer share one data source. The simplified architecture is shown in the figure 8.1.

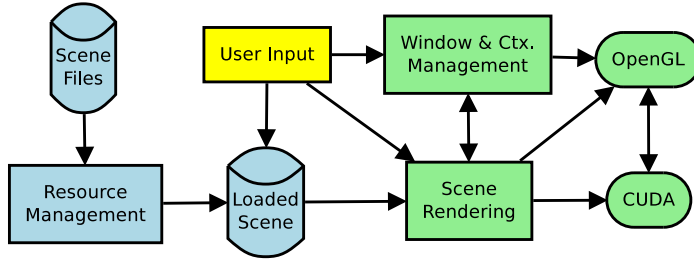


Figure 8.1: The application is designed using the Model-View-Controller (MVC) pattern. Blue modules work as the model, user input controls the scene and the rendering, green modules work as the view subsystem.

8.2 Code Structure

We organize our code into a couple of smaller modules. The most of the functionality is contained by the standalone `core` library. The application with a simple GUI is separated and linked against the `core` library.

The core library is made of several components. The components are listed in the table 8.1. GLSL shader kernels are separated from the C/C++ code and are loaded on-line by the `io` module. Particular components are described in the following sections.

module	functionality
gfx	abstract rendering devices, helper classes for the graphics subsystem
io	resource loading and storing
ism	Imperfect Shadow Maps based renderer
math	a simple mathematical library
ogl	OpenGL renderer and supporting classes
pt	path-tracing code
scn	scene data structures and a scene manager
util	timers, random number generators and utility classes

Table 8.1: Components enclosed by the core library

8.2.1 Resource Loading and Management

Resource management is done by the `io` module. The `ResourceManager` class takes care of the resource loading and creates a cache for the loaded resources. We use abstract factories for the resource loading itself. So it is possible to extend the functionality by adding a new input/output back-ends. The `DevILImageLoader` class encapsulates the DevIL image manipulation library. Scene loading is encapsulated by the `AiSceneLoader` class objects, which uses the Assimp library. Both libraries were described in the chapter 7.

All the resources are reference counted. We use the `boost::shared_ptr` smart pointers [Boo12] to achieve the desired behaviour. We use the reference counting especially to avoid data duplication. For example, if two objects in the scene share the same geometry, data is not duplicated, as both objects hold only the reference to the actual data. Also the

loading is done only once, as the `ResourceManager` class object caches the result of the first loading procedure.

8.2.2 Scene Management

Scene management code is located in the `scn` module. After a successful load, scene object references are stored inside the `Scene` class object. The `Scene` class contains a hierarchical representation of the scene, called scene graph. The `Scene` class also offers various scene object queries. For example, to accelerate the rendering, we use the viewing frustum culling technique described in the section 3.3. We implement VFC as a scene query. The renderer executes the query on the scene and iterates through returned objects. A simple code sample is listed in the figure 8.2.

```
Scene::Query * vfq = scene->query(viewport->frustum(), worldToCamMat);
while(vfq->hasNextObj()) {
    processObject(vfq->nextObj());
}
scene->releaseQuery(vfq);
```

Figure 8.2: Example usage of a view frustum query in our implementation

The `Scene` class also works as an interface between the model and view subsystems in the MVC design. The class provides event generation system to notify attached views (renderers in our case). Controlling particular scene objects can be done using the `SNController` and derived classes.

8.2.3 Rendering Subsystem

There are several rendering back-ends in our application. All rendering classes inherit the abstract `SceneRenderer` class, located in the `gfx` module. The rendering method can be switched online (see appendix C for more details). When a new renderer is created, it attaches itself to the `Scene` object and reads all the scene objects. It also listens to the scene events and alters its state when the scene is changed (for example when a new scene is loaded from a file).

The most important is the ISM renderer, which lies in the `ism` module (plus shaders). We deeply describe the workflow of this module in the section 8.3. The `ogl` module contains basic OpenGL based renderer, which we use for debugging. It only calculates the direct illumination.

8.2.4 Path-Tracer

Our simple reference path-tracer is placed in the `pt` module. We use kd-tree [Ben75] as an acceleration data structure. The tree is rebuilt on every scene change. That practically limits the usage of this renderer to geometry-static scenes. Lights and cameras can be freely

moved around the scene. These preferences are sufficient for us, as we use path-tracing only as a reference solution in our quality analysis, see the section 9.4 for more details.

Path-tracing itself is done on the GPU, accelerated by CUDA. We use limited stack traversal algorithm known as short-stack [HSHH07]. In one pass, for each ray bounce, one shadow ray and one reflection ray is generated. Incoming radiance is sampled with respect to the BRDF times $\cos(\theta)$, see section 2.3. Successive frames use a different random number generator seed. The result is accumulated in a frame buffer with a floating point pixel format.

We compare the camera matrix from the previous frame to the new one to detect the movement. If the camera position does not change, image converges to the correct value after some time. The advantage is that user can immediately see the low quality noisy image after he moves the camera.

8.2.5 Graphical User Interface

As said before, we use Qt libraries to create our GUI. We have separated the GUI code from the rendering core. The GUI is a part of the main application, called `qtgui`. The application creates two windows, one displays the rendered result, the second contains the UI. Our UI allows the user to change several rendering settings online (or select the rendering back-end). The GUI look is shown in the appendix C.

8.2.6 Supporting Code and Dependencies

The implementation contains many small classes which support the main code. The utility code is located in the `util` module. The table 8.2 shows all used libraries and the version we have used. Some of the dependencies are shipped with the application to simplify the building process. Especially on Windows platforms where the installation of particular libraries would be uncomfortable. The listed OpenGL version denotes the "functionality version", OpenGL vendors typically distribute their implementation with the graphics driver software or their graphics SDK. Thus the versioning can differ. For more details of the installation process, see the appendix C. The detailed directory listings are included in the appendix D.

Library	Version	Installation
Assimp	2.0	included
boost	1.46.1	required
CUDA	4.0	required
DevIL	1.7.8	included (WIN)
GLEW	1.7.0	included (WIN)
OpenGL	(4.0)	required
Qt	4.7.4	required
v8	3.8.9.6	included

Table 8.2: All library dependencies

8.3 Rendering Pipeline

In this section we describe all stages of the implemented ISM renderer in detail. The implementation is based upon the ISM renderer described in the section 4.11. The figure 8.3 shows the connections between particular rendering stages. The scene is rasterized from the view of a camera, the result is stored in a geometry buffer (g-buffer). The g-buffer is read by a couple of subsequent stages. The SSAO stages use the depth and normal component to create a screen space ambient occlusion approximation. The discontinuity detection stage reads the depth and normal component, and creates a bit map which indicates edges in the rendered g-buffer. The g-buffer is also shuffled to form defined number of sub-buffers. The g-buffer is also shuffled to form defined number of sub-buffers.

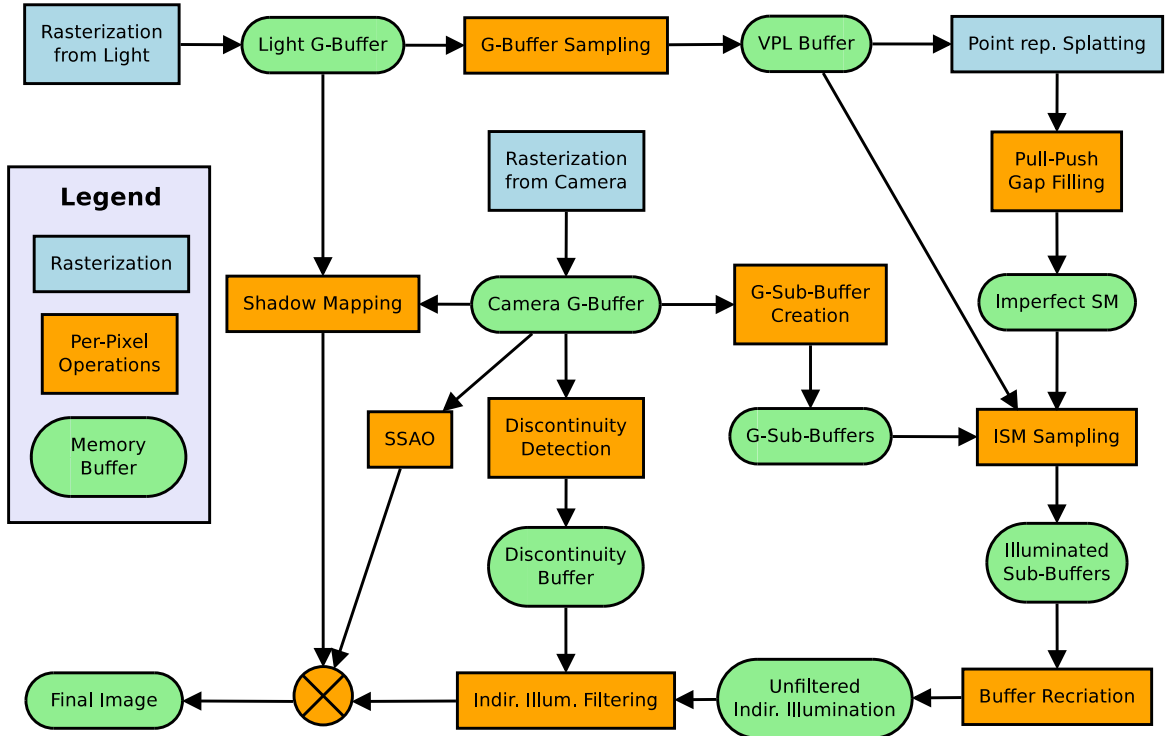


Figure 8.3: The architecture of our ISM implementation

From the view of a light, a geometry buffer is created as well. The light g-buffer is then sampled and virtual point lights are created. The VPLs stored in a buffer are read by a subsequent ISM creation stages. The point representation of the scene is splatted, creating a ISM texture. The ISM texture is refined by a pull-push algorithm which tries to fill the gaps in a splatted image. The pull-push stage outputs the final imperfect shadow map.

The pipeline continues in the indirect illumination evaluation by executing the ISM sampling stage. This stage produces illuminated sub-buffers. The illuminated sub-buffers are shuffled again and one illuminated buffer is created. The indirect illumination is filtered using a geometry-aware kernel.

In the last stage the direct illumination is combined with the indirect illumination, and SSAO is added. It is obvious that most of the stages are computed on the pixel domain. We discuss the effects of this characteristic in the section 9.3.

8.3.1 Preprocessing

Our application can generate point representation in two ways. The original algorithm, described in the paper [RGK⁺08], generates point samples over the geometry off-line, in the preprocessing stage. We added the possibility of on-line generation which, is described in the section 8.3.6.

If the on-line sampling is disabled, the point representation is created preliminary to the rendering loop. The point representation should cover the scene uniformly. We work only with a boundary representation of the scene, stored purely as triangles. So the point density should be constant over the whole boundary area. To achieve this, we create a probability function with values proportional to the area of triangles, and sample triangles with respect to their probability. The probability p_i for the i -th triangle is calculated as a fraction A_i/A . Where A_i is the area of i -th triangle and A is the sum of all triangle areas.

The sampling uses a search in a cumulative distribution function which we create from our triangle probability function. To place a sample, we need three random numbers. The first is used to find a triangle in the cumulative distribution function. From the second and third number, we generate barycentric coordinates. The code shown in the figure 8.4 uses the binary search `lower_bound` to find a triangle and generates the barycentric coordinates. The created point samples are stored in a vertex array, next to the sampled geometric data.

```
Vec3f rvec = random_seq.next();
float * ar = lower_bound(cdf, cdf + N, rvec.z * A);
if(rvec.x + rvec.y > 1.0f)
{
    rvec.x = 1.0f - rvec.x;
    rvec.y = 1.0f - rvec.y;
}
Vec3f barc(rvec.x, rvec.y, 1.0f - rvec.x - rvec.y);
uint tri_id = ar - cdf;
```

Figure 8.4: Random placement of samples using binary search in the cumulative distribution function; barycentric coordinates are generated using a method presented by Glassner [Gla93].

8.3.2 Geometry Buffer Stage

Our implementation uses a deferred rendering approach. We use a three-component frame-buffer to store the depth, normal and material properties, as it is shown in the table 3.1. The normal is compressed into two sixteen-bit numbers, using the sphere-map encoding [Mit09]. We use textures without multisampling as our render targets, so our implementation suffers from the aliasing issues.

The rendering is accelerated using the view frustum culling based on axis-aligned bounding boxes of the scene objects. We do not use any acceleration data structures, like bounding

volume hierarchy (BVH), the bounding boxes are stored in a linear list. Also no kind of sorting or grouping is used during the rendering process.

The scene is stored in a memory of a GPU using OpenGL vertex arrays. The implementation supports textured and coloured materials. To reduce the shader complexity, we store the colour in a one-pixel texture for the objects that do not have regular textures. As discussed in the section 8.2.1, all graphical resources are reference-counted and shared within the scene objects.

8.3.3 Shadow Map Generation

The direct illumination is computed using the classic shadow mapping approach as was described in the section 3.1. For each light, we render the shadow maps to six faces of a cube map, so our implementation supports omnidirectional lights. We render each face in a separate pass, so we can use the viewing frustum culling (VFC) to minimize the number of primitives drawn. The default resolution of a shadow map is set to 1024×1024 pixels.

To avoid self-shadowing problems, we use a small offset during the shadow evaluation. The offset value is derived from the extent of the scene. Our application can take a benefit from the hardware accelerated shadow map filtering, as we enable the comparison mode for shadow samplers in our shaders. We do not use any soft shadow mapping algorithm, so our implementation supports only point lights.

8.3.4 Reflective Shadow Map Generation

We separate the reflective shadow map generation from the classic shadow map stage. We also render reflective shadow maps to faces of a cube map, but we use a lower resolution than in the classic shadow map stage. As the lower resolution is sufficient for the virtual point light (VPL) sampling, we can spare the memory space and bandwidth. Our reflective shadow maps use the same frame buffer format as we use in the deferred rendering stages. However, the CUDA version 4.0 we use does not support sharing native depth buffers. We have solved this issue by creating one additional render target with a four-byte float format where we duplicate the depth values. In default, we render the reflective shadow map in the resolution of 256×256 pixels.

8.3.5 VPL Creation Stage

In contrast to the preceding stages, the VPL creation is done using the CUDA C language. A quasi-random sequence is used to generate the sampling coordinates. We use the Halton sequence [Hal64] which we pre-compute and store in the texture memory of a GPU. The precomputed sequence should be long enough to provide sufficient number of quasi-random numbers.

The reflective shadow map samples are fetched from the cube-mapped g-buffer. For each sample, we evaluate the incoming radiance and calculate the future VPL intensity. The incoming radiance is evaluated considering only the actually processed "primary" light. Our implementation offers two sampling approaches. The first and simpler approach accepts all samples, so the VPLs are evenly distributed over the light field of view (which can be full

sphere in the case of omnidirectional light). The second approach uses the rejection sampling and accepts the future VPLs with respect to their estimated importance.

The importance metric combines two factors. The first importance factor is the intensity of the future VPL. It is wise to accept VPLs with a higher intensity, as there is a higher chance that they will contribute to the final illumination of the scene. The second factor is based on the position of the future VPL. Every VPL is treated as a hemispherical light. So, if the VPL is positioned and oriented in a way that it cannot illuminate the visible part of a scene, it is completely useless. The relationship between the position of the future VPL and the visible part of a scene is hard to evaluate precisely. We select a random subset of scene points visible from a camera, and evaluate the visibility from the future VPL only for this subset of samples. The visible points are extracted from the geometry buffer created during the scene rasterization from the view of the camera.

To keep the sampling pattern unbiased, we divide the intensities of the accepted samples by the total number of samples taken (including the rejected samples). As the sampling runs in many threads, we use parallel prefix sum to calculate the total number of samples taken by all threads.

The accepted VPLs are stored in a memory buffer on the GPU. For each VPL, we have to store its intensity and a world-space position and orientation. The straightforward approach is to store the position as a three-dimensional vector. The orientation can be stored in various ways. The spherical coordinate system is probably the most data-saving approach, since it requires only two-dimensional vector of two angles. However, following stages have to perform a transformation to the coordinate system of the VPL. This operation is performed many times as there is a high number of fragments that calculate the contribution of each VPL.

For spherical coordinates, the usage of goniometric functions during the coordinate system conversion is a must. However, the frequent usage of goniometric functions can introduce a performance drop. So, we have chosen more data consuming encoding of a position and orientation of the VPLs that does not require the usage of goniometric functions in the later stages. We encode the VPL position and orientation as a classic OpenGL transformation matrix. As all transformations we need can be encoded using only the first three rows, the fourth row would always be equal to $(0\ 0\ 0\ 1)$. Using this fact, we can store the intensity of the VPL in the fourth row.

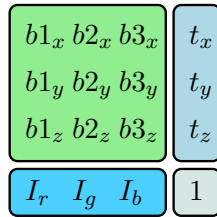


Figure 8.5: We store each VPL in a 4×4 floating point matrix. The intensity vector I is stored in the fourth row of the matrix.

Using the VPL encoding shown in the figure 8.5, we can store each VPL as a sixteen-float matrix. If the VPL transformation matrix is required, then only the fourth row has to be

reseted back to the $(0\ 0\ 0\ 1)$ form. There are no additional numeric operations. Also all the VPL matrices follow the 64 B alignment that helps the GPU architecture to maximize the memory throughput.

8.3.6 ISM Splatting

The fundamental part of the rendering process is the imperfect shadow map creation. The ISMs are stored in one large square texture and they are created in one rendering pass. We create one ISM for each virtual light, so it is efficient to have the number of VPLs $N = M^2$ where M is the number of ISMs in one row of the large texture. In practice, we choose the number of VPLs N equal to some even power of two.

The splatting algorithm has to read (or generate) the point representation of a scene and distribute the future splats uniformly over all ISMs. For each point sample, we use a quasi-random number to designate a targeted VPL/ISM. As all VPLs are stored as a transformation matrix with embedded intensity, the splatting algorithm has to clear the intensity part and multiply the actual point coordinates with the matrix to obtain the VPL-space coordinates. We have chosen the same projection (parabolic mapping) as presented in the original publication [RGS09].

The diameter of a splat is evaluated using a simple formula: $d = k/l$, where d is the diameter, k is the precomputed constant and l is a distance between the point splat and the targeted VPL. The precomputed constant k controls the scale of all splats. It is estimated from the scene extent, the number of point samples and from the resolution of the ISM texture. The diameter d is then limited by a maximum value to avoid the creation of too large samples from points that are too close to the VPL.

We have implemented the splatting in a couple of ways. The first approach uses the samples precomputed in the preprocessing stage and renders their splats using OpenGL point primitives. The splat diameter d is used to control the size of rasterized point. The second implemented method uses OpenGL geometry shaders to generate quads instead of points. The quads have the same size as the corresponding point primitive would have. This method was implemented mainly to explore the performance of a quad rasterization compared to a point rasterization.

The important innovation of the ISM algorithm is the online tessellation approach. We have used the tessellation unit, which is available in recent GPUs, to generate the point representation during the rasterization process. The tessellation unit is primarily used to generate a finer geometry by subdividing triangles or quads [SA11]. But the OpenGL offers also so-called "point mode" tessellation that generates point primitives where the finer geometry vertices would be. The tessellation pattern can be seen in the figure 7.5.

In this method, we do not use the preprocessed point representation. Instead, we generate points directly from the scene triangles. The only precomputed information we use is the sum of the areas of all triangles. For each triangle, we determine the number of wanted point samples from its area. According to the tessellation pattern, the number of unique vertices N , generated in a tessellation level L , is given by the equation 8.1.

$$N = \frac{3(L+1)^2}{4} \quad (8.1)$$

If we invert the relationship between N and L , we can get the tessellation level L as a function of the number of generated points N , that is shown in the equation 8.2.

$$L = 2\sqrt{\frac{N}{3}} - 1 \quad (8.2)$$

The second solution of the quadratic equation is skipped as the number of generated points N is always non-negative. Both equations hold only for odd tessellation levels as even levels create a different tessellation pattern. As the number of generated points does not have to be precise, we generalize the formula for all levels.

If the scene objects deform, then the total area of the scene surfaces can vary. We can use OpenGL queries to retrieve the number of point samples that were generated. This information can be used as a feedback and the precomputed area sum can be altered.

A tessellation control shader uses the formula above and commands the hardware tessellation unit to create the desired number of point samples. The rest of the process is similar to the previous approaches, geometry shader randomly selects a targeted ISM, calculates the projection and the splat size.

The point representation splatting can naturally create holes in the future ISM. To fill the holes, we use the pull-push algorithm [MKC07]. The pull-push stage is implemented in the CUDA C language and shares the ISM texture with the OpenGL environment. The difference between the ISM with and without pull-push post processing is shown in the figure 9.5.

8.3.7 Discontinuity Detection

As the indirect illumination is evaluated in a stochastic fashion, it usually requires additional filtering to remove the noise. The filtering usually does not create any distinct artifacts, as the indirect illumination often varies smoothly. We use a Gaussian kernel combined with a discontinuity detection to avoid mixing of illumination of distant parts of the scene.

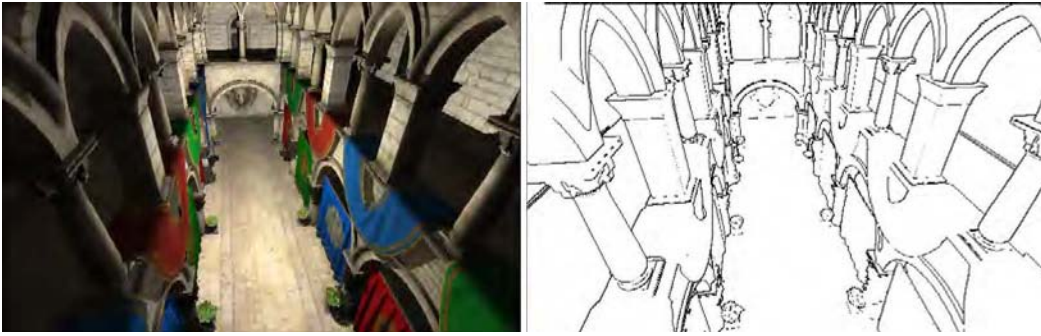


Figure 8.6: The Crytek-Sponza scene rendered with our implementation (left) and the discontinuity buffer calculated for the same view (right)

The discontinuity detection is separated from the filtering stage. For each pixel in a geometry buffer, we evaluate the difference between the stored depths and normals of neighbouring pixels. If the difference is bigger than a configured threshold, the discontinuity flag

is written into a discontinuity buffer. An example scene with its discontinuity buffer is shown in the figure 8.6. Similar approach was presented by Segovia et al. in 2006 [SIP06].

8.3.8 Indirect Illumination Stage

When the VPL buffer and imperfect shadow maps are ready, the indirect illumination can be evaluated. The exact task of this stage is to calculate the visibility and evaluate the irradiance for all visible parts of the scene with respect to all VPLs. For each pixel in a g-buffer we evaluate the indirect illumination by a stochastic sampling of the VPLs. Due to memory bandwidth limits, it is impossible to evaluate the contribution of all VPLs for all pixels when we want to keep the real-time performance. The idea used by Ritschel et al. [RGS09] was to sample a random subset of VPLs in each pixel, and filter the result using the geometry-aware kernel as presented in the previous section.

However, the random sampling creates a highly random memory access pattern by its nature and creates huge performance problems. This phenomenon can be partially suppressed by grouping pixels that will evaluate the same subset of VPLs. It is because they will access the similar memory region when evaluating the visibility with the ISM sampling. One solution is to create a number of geometry sub-buffers and evaluate the same subset of VPLs in all pixels within a sub-buffer [SIP06].

8.3.8.1 Sub-Buffer Generation Stage

Sub-buffers are created by a re-organization of an original g-buffer. The next stage reads the normal buffer and the depth buffer so the reorganization is performed only on those two buffers (the third g-buffer with a material properties is omitted from this stage). We store the created sub-buffers in a Multiple Render Targets frame buffer of the same size as the original g-buffers. Sub-buffers are stored side-by-side in a $N \times N$ array. Each sub-buffer is formed by a selection of every N -th pixel. We perform the operation in a GLSL fragment shader which is executed for all pixels in the sub-buffers. The result of this stage is shown in the figure 8.7.

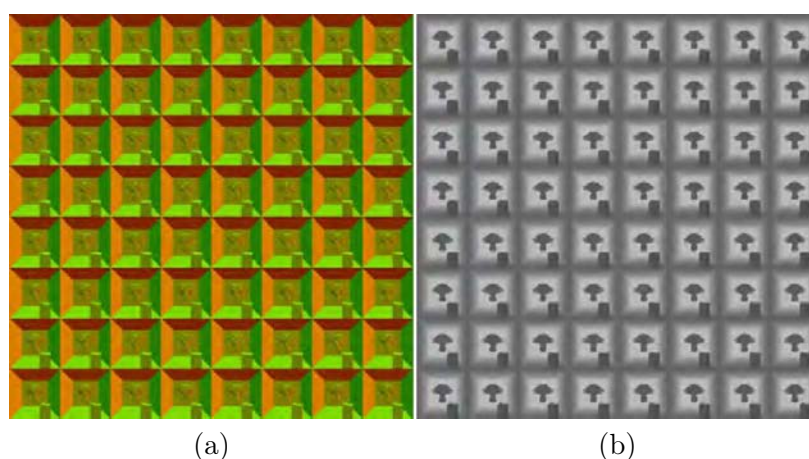


Figure 8.7: Normal (a) and depth (b) sub-buffers organized in a 8×8 pattern

8.3.8.2 VPL Sampling

The indirect illumination is evaluated over the sub-buffers created in the previous stage. As for the previous stage, we use the GLSL fragment shaders to perform the evaluation. For each pixel on the screen, the algorithm firstly identifies the coordinates of a sub-buffer in which the pixel lies. The sub-buffer coordinates are used as a random number generator seed. The random sequence is obtained from a precomputed two-dimensional random texture by using the sub-buffer coordinates for the texture lookup.

The random sequence is used to select the subset of VPLs whose contribution will be evaluated. We evaluate the contribution of sixteen VPLs in each pixel by default. For each VPL, the algorithm fetches its matrix (the format of VPL encoding was described in the section 8.3.5), translates the actual pixel position to the coordinates of a VPL and performs a ISM lookup. If the VPL is visible from the actual pixel, then the irradiance is calculated. The accumulated irradiance over all sampled VPLs is stored as a result of this stage.

The imperfect shadow maps suffer from the same drawbacks as classic shadow maps do. The biggest problem, aside the imperfection, is the self-shadowing caused by the low resolution of an ISM. Here we use the same biasing approach as is known from the classic shadow mapping. However, the resolution of a ISM is usually much lower than for a classic shadow map. So the bias should be set relatively high, we use 15 % from the depth by default.

As the indirect illumination is evaluated over the sub-buffers, we have to recreate the original one-buffer layout. This is done by de-shuffling the sub-buffers in a process opposite to the sub-buffer creation described in the section 8.3.8.1.

8.3.8.3 Indirect Illumination Filtering

The unfiltered irradiance can be seen in the figure 8.8. The result is rather noisy, also the sub-buffer pattern is obvious from the image. We use a Gaussian, geometry aware kernel to filter the noise out. The kernel reads a discontinuity buffer created in a preceding stage, and stops the accumulation of values in directions where the discontinuity is found. Problems

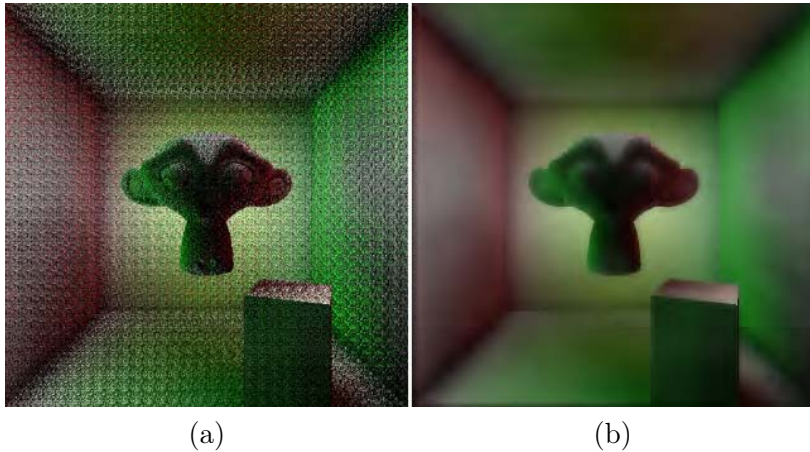


Figure 8.8: Unfiltered (a) and filtered (b) irradiance

can occur with pixels that are surrounded by discontinuities. In such places, there are not enough close pixels that can be used to filter the noise out. This usually creates small notable artifacts in places where discontinuities are close to each other.

8.3.9 SSAO Stage

As discussed before, the ISMs cannot handle local occlusions correctly due to their limited resolution and to the high biasing. On the other hand, an SSAO algorithm can simulate small local effects in the lighting relatively well. We have implemented a simple SSAO algorithm to detect the local occlusions and we combine the result of the SSAO stage with the indirect illumination calculated with the help of ISMs.



Figure 8.9: Part of the Crytek-Sponza scene illuminated only by indirect illumination; an image rendered without (a) and with (b) the SSAO; the difference is especially notable around the corners.

We estimate the SSAO for each pixel by a blocker search in its neighbourhood. We use a random two-dimensional texture to enforce the stochastic SSAO evaluation. For each pixel we march in four orthogonal directions and take four samples per direction by default. The result is perceptually plausible and does not require any additional filtering.

8.3.10 Direct Illumination and Final Composition

The final stage has to combine the indirect illumination with direct one, and apply the material properties of the scene. We use classic shadow mapping to detect a direct shadow. Our implementation supports only point lights since there is no soft shadowing implemented. To avoid self-shadowing, we use 4 % of the stored depth value as a bias with the shadow maps for the direct lighting by default. This stage is implemented in a GLSL fragment shader and the output is sent directly to the default frame buffer.

8.3.11 Time Measurement

We require precise time measurement for the graphic operations in our application. Using classic CPU timers is not efficient since it requires stalling the CPU and waiting for the

GPU operations to become finished. To avoid stalls, the timers have to work asynchronously (CPU code should not be blocked while the measurement is in progress). OpenGL offers such functionality with Timer Queries. The CPU code starts a query by calling the `glBeginQuery` function. The query is queued into the OpenGL command list, so the function returns instantly. The query is stopped with the `glEndQuery` call which is also asynchronous, so the function returns immediately.

The query issues a time measurement of OpenGL commands executed within the query boundaries. As the whole query mechanism works asynchronously, the query result does not have to be available right after the query was stopped with the `glEndQuery` function call. So in each frame, we read the results of queries that were issued during the last frame. That should minimize the stalls since all drawing commands from the last frame should be done.

Chapter 9

Results

We have tested our implementation in various conditions and on various hardware setups. We have selected five different scenes as a sample data set. The selected scenes vary in geometrical and lighting complexities. We have also created synthetic scenes to test the algorithm in extreme conditions. We analyse the quality of rendered images and compare the results with our reference path-traced solutions.

9.1 Tested Scenes

Our testing scenes are listed in the table 9.1 and 9.2. The U-shaped scene was created to test the algorithm in hard lighting conditions. The scene has a U-like shape and there is no direct light path between the camera and the light. On the other hand, the U-shaped scene is geometrically trivial. The Monkey Box scene is our second synthetic scene. The Monkey Box scene consists of a box with strongly coloured side faces and two meshes¹. There are no big geometrical complexities in the Monkey Box scene, also the lighting conditions are simple.

	U-Shape	Monkey Box	Sibenik Cathedral
No. triangles	76	7136	75284
Geometry	simple	simple	medium
Lighting	hard	simple	medium

Table 9.1: Tested scenes with the number of triangles, geometrical and lighting complexities

The Sibenik scene is a well-known model of a well-known cathedral located in Sibenik, Croatia. The scene has medium geometrical and lighting complexities. The Crytek Sponza

¹The monkey mesh is generated by Blender 3D content creation suite <<http://www.blender.org/>>

scene is a remake of Atrium Sponza Palace model, originally done by Marko Dabrovic. The scene has a relatively high geometrical and lighting complexity. The last chosen scene is the Conference Room model. It has relatively high triangle count, but the lighting conditions are not as complex as in the Conference scene. The Sibenik Cathedral, Crytek-Sponza and Conference Room scenes were downloaded from the McGuire Graphics Data pack, available at <http://graphics.cs.williams.edu/data/>.

	Crytek Sponza	Conference Room
No. triangles	262267	331179
Geometry	hard	hard
Lighting	hard	medium

Table 9.2: Tested scenes with the number of triangles, geometrical and lighting complexities

9.2 Used Hardware Setups

We have tested our implementation on three different hardware setups. As said before, our implementation requires CUDA-enabled graphics card. Also we require OpenGL version 4.0 features. This forces us to choose only from nVidia Fermi and newer GPUs [NVI09].

	HW560	HW580	HW470
GPU	GF GTX 560 Ti	GF GTX 580	GF GTX 470
CUDA Cores	384	512	448
GPU Mem.	1 GB, 256 b	3 GB, 384 b	1280 MB, 320 b
CPU	C2D E8300	Core i7 2600K	Core i7 950
RAM	4 GB	16 GB	12 GB
OS	Linux 3, 64 b	Win7, 64 b	Win7, 64 b

Table 9.3: Used hardware setups

9.3 Running Time Analysis

In this section we would like to present the performance of our implementation. We vary some of the render parameters and discuss their influence on the algorithm running time. If it is not explicitly mentioned, the measurement was done with these parameters: resolution 512×512 px, shadow map resolution 1024×1024 px, 10^6 point samples, 1024 VPLs, ISM resolution 2048×2048 px, runtime tessellation enabled, pull-push enabled and SSAO enabled.

The presented values are the average of ten repeated measurements. All measurements were done using the OpenGL timer queries as discussed in the section 8.3.11.

Table 9.4 show the rendering time breakdown for the standard settings. It is obvious that for most setups, rendering time is evenly distributed between the particular stages. Higher times for the texture intensive operations on the HW580 and HW470 setups is probably caused by a lower number of texture units accessible on their GPUs [NVI09].

Scene	FPS	D+Mix	I.Flt	II	ISM	Cam.G	SM	SSAO	VPLS	Sub-B
HW560, 512×512 px										
U-Shape	100	0.092	1.3	1.0	6.4	0.10	0.17	0.33	0.28	0.24
Monkey	34	0.130	1.3	1.1	26.0	0.17	0.33	0.30	0.34	0.24
Sibenik	15	0.130	1.3	1.1	17.0	8.00	20.00	0.34	20.00	0.25
Sponza	33	0.110	1.3	1.1	12.0	1.80	6.10	0.52	7.40	0.25
Confer.	52	0.130	1.3	1.1	10.0	0.56	3.10	0.34	2.60	0.24
HW580, 512×512 px										
U-Shape	85	0.290	3.5	3.3	3.4	0.21	0.16	0.90	0.64	0.30
Monkey	50	0.300	3.6	3.5	11.0	0.13	0.20	0.88	0.46	0.30
Sibenik	37	0.300	3.5	3.8	7.8	2.00	3.90	0.89	5.10	0.31
Sponza	54	0.290	3.5	3.8	5.9	0.60	1.80	0.91	2.20	0.31
Confer.	60	0.300	3.5	3.9	5.1	0.32	1.40	0.89	1.70	0.31
HW470, 512×512 px										
U-Shape	61	0.390	5.3	4.9	4.5	0.13	0.17	1.3	0.49	0.42
Monkey	37	0.420	5.3	5.2	15.0	0.16	0.25	1.3	0.50	0.44
Sibenik	28	0.410	5.2	5.6	10.0	2.50	5.30	1.3	6.00	0.44
Sponza	39	0.400	5.2	5.6	7.9	0.82	2.70	1.3	2.60	0.44
Confer.	43	0.420	5.2	5.8	6.7	0.43	2.10	1.3	2.40	0.44

Table 9.4: Breakdown of the rendering time for all tested scenes for all hardware setups; the table shows frames per second (FPS), direct illumination and composite time (D+Mix), indirect illumination filtering time (I.Flt), indirect illumination sampling time (II), imperfect shadow map creation time (ISM), camera geometry buffer time (Cam.G), shadow map generation time (SM), screen space ambient occlusion time (SSAO), VPL generation time (VPLS) and geometry sub-buffer creation time (Sub-B). All times are in milliseconds. All scenes were rendered in the resolution of 512×512 pixels.

Rendering on higher resolution slows the pixel-intensive operations as is shown in the table 9.5. Times for operations like SSAO or indirect illumination filtering rise as expected. Other stages, like ISM splatting, keep their performance.

Our next measurement shows the difference between the static and dynamic tessellation running times. The dynamic tessellation is faster in the most cases. It is caused by a higher memory bandwidth which is required by the static tessellation. In all cases the number of point samples is much higher than the number of all triangles in the scene. It means that a larger data flow is required when the precomputed point samples are drawn. The table 9.6 shows the results for all tested hardware setups.

The table 9.7 shows how geometry sub-buffer layouts influence the VPL sampling time. The 1×1 layout means no sub-buffer creation. It involves the evaluation of different random

Scene	FPS	D+Mix	I.Flt	II	ISM	Cam.G	SM	SSAO	VPLS	Sub-B
HW560, 1280×720 px										
U-Shape	62	0.29	4.3	3.5	6.4	0.19	0.15	1.2	0.28	1.05
Monkey	27	0.31	4.4	3.6	26.0	0.20	0.22	1.0	0.33	1.05
Sibenik	13	0.34	4.4	3.8	17.0	8.30	20.00	1.5	21.00	1.24
Sponza	27	0.32	4.3	3.7	12.0	2.00	5.80	1.9	7.40	1.32
Confer.	38	0.34	4.3	3.8	10.0	0.73	2.90	1.0	2.60	1.14
HW580, 1280×720 px										
U-Shape	34	0.88	12.0	12.0	3.4	0.18	0.11	3.1	0.52	1.07
Monkey	26	0.85	12.0	12.0	12.0	0.16	0.15	3.0	0.57	1.07
Sibenik	22	0.90	12.0	12.0	7.8	2.10	3.90	3.1	5.10	1.10
Sponza	27	0.90	12.0	12.0	6.0	0.77	1.70	3.2	2.20	1.10
Confer.	28	0.93	12.0	13.0	5.3	0.43	1.30	3.1	1.70	1.10
HW470, 1280×720 px										
U-Shape	23	1.30	18.0	17.0	4.5	0.23	0.17	4.5	0.49	1.52
Monkey	18	1.30	18.0	18.0	15.0	0.26	0.23	4.4	0.53	1.53
Sibenik	16	1.30	18.0	18.0	10.0	2.40	5.30	4.6	6.10	1.57
Sponza	19	1.30	18.0	18.0	8.0	1.10	2.40	4.6	2.60	1.56
Confer.	19	1.40	18.0	20.0	6.7	0.60	2.00	4.5	2.40	1.55

Table 9.5: Breakdown of the rendering time for all hardware setups; all times are in milliseconds. The resolution was magnified to 1280×720 pixels. The legend is the same as for the table 9.4.

	HW560			HW580			HW470		
Scene	T_s [ms]	T_d [ms]	S [–]	T_s [ms]	T_d [ms]	S [–]	T_s [ms]	T_d [ms]	S [–]
U-Shape	18	6.4	2.81	8.3	3.4	2.44	10	4.5	2.22
Monkey	28	26.0	1.08	12.0	11.0	1.09	14	15.0	0.93
Sibenik	18	17.0	1.06	8.0	7.8	1.03	10	10.0	1.00
Sponza	21	12.0	1.75	9.1	5.9	1.54	11	7.9	1.39
Confer.	19	10.0	1.90	8.5	5.1	1.67	11	6.7	1.64

Table 9.6: Comparison between the point sample generation methods; for each hardware setup we measure the time of the ISM stage when using pre-processed point samples T_s and dynamically generated point samples T_d . The speedup S shows how many times is the dynamic method faster.

subset of VPLs for each pixel on the screen. This approach is markedly slower as we have expected. On the other hand, grouping pixels in sub-buffers increases the performance of the indirect illumination stage rapidly.

The following table 9.8 shows the difference between the splatting times when using points or quads. The quads method uses a geometry shader to create a quad instead of a point. This measurement fulfills our expectations, the quad method is slower since there is a overhead of creating new vertices.

Scn.	1×1	2×2	4×4	8×8	16×16	32×32
U-Shape	27	6.6	1.3	1.1	1.0	1.0
Monkey	27	6.6	1.4	1.1	1.1	1.1
Sibenik	27	6.6	1.6	1.2	1.1	1.1
Sponza	27	6.6	1.5	1.2	1.1	1.1
Conference	27	6.6	1.7	1.2	1.2	1.1

Table 9.7: Different geometry sub-buffer configurations and the running time of the indirect illumination stage; all times are in milliseconds.

Scene	T_{pt} [ms]	T_q [ms]	S [–]
U-Shape	18	25	1.39
Monkey	28	37	1.32
Sibenik	18	26	1.44
Sponza	20	29	1.45
Conference	19	28	1.47

Table 9.8: Comparison between the splatting time of points T_{pt} and quads T_q ; the speedup S shows how many times is the quad splatting method slower.

9.4 Quality Analysis

In this section we discuss how certain parameters of the ISM algorithm influence the quality of the rendered result. The number of point samples and the VPL count are the most important parameters to investigate. The importance of the pull-push state is also discussed. We show the difference between our implementation and the reference path-tracer as well.

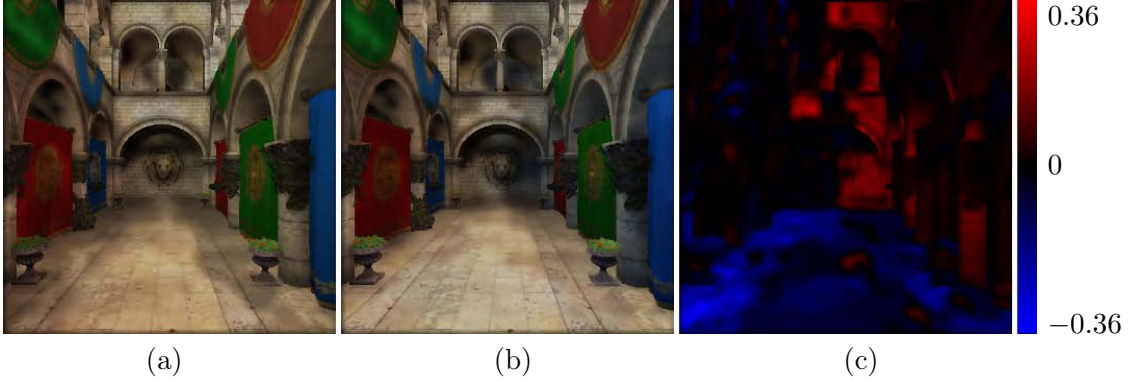


Figure 9.1: Indirect illumination of the Crytek-Sponza scene; both images were rendered with a close light source position, the number of VPLs was set to 16. The low number of VPLs caused a strong flickering in the illumination. The image (c) shows the difference between the image (a) and (b).

The figure 9.1 shows how low VPL count degrade the quality of the indirect illumination. It also negatively influences the temporal coherence. Both rendered images in the figure 9.1 show only the indirect illumination of the Crytek-Sponza scene. The light source was slightly

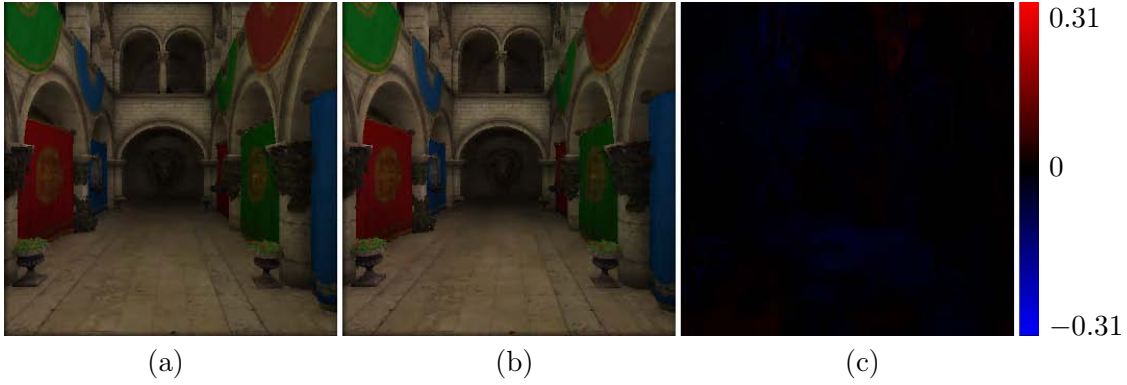


Figure 9.2: Two rendered images of the Crytek-Sponza scene with a close light source positions; the number of VPLs was set to 1024. The difference in the indirect illumination is only small in contrast to the results shown in the figure 9.1. This figure shows the correct behaviour.

moved when rendering the second image. The rest of the scene remained completely static. The indirect illumination is expected to vary only smoothly in such scenarios. However, the insufficient number of VPL samples caused distinct changes in the indirect illumination.

A more correct behaviour of the indirect illumination is shown in the figure 9.2 where we set the VPL count to 1024. Both images were rendered with the same light positions as in the figure 9.1. The difference image indicates that the illumination has changed only gently as expected.

Images 9.3 and 9.4 show how the number of point samples influences the shadow map quality. The low number of point samples usually increases the imperfection of the generated ISM. The situation gets ever worse when the pull-push phase is omitted. This scenario is shown in the figure 9.4. The difference between the ISM with and without the pull-push refinement is shown in the figure 9.5.

The figure 9.6 shows a comparison of our ISM implementation to the reference path-tracer. The ISM algorithm generally produces a little bit brighter images. That is caused by light leaks that emerge from the imperfection of the shadow maps. As we do not use any anti-aliasing method, strong differences around the object and shadow contours can appear. The figure 9.7 shows the difference between the reference image and the ISM rendered image of the U-shaped scene. The ISM algorithm suffers from the light leaks, as discussed before, it also fails to capture the details of the illumination on the farther wall. That is caused by the sampling approach which places most of the VPLs around the primary light source, so the farther parts of the scene are sampled insufficiently.

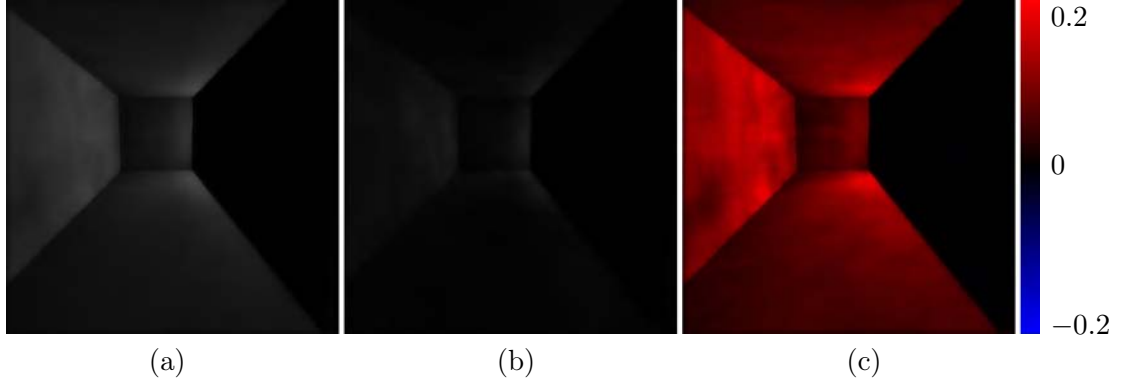


Figure 9.3: Comparison of two images of the U-shaped scene with various number of point samples used; the camera look is oriented at the part of the scene that is completely shadowed. The light is placed behind the wall. The image (a) is rendered using only 10^5 samples, the image (b) is rendered using 10^6 point samples. The light "leaks" through the wall since the low number of samples creates holes in the ISM.

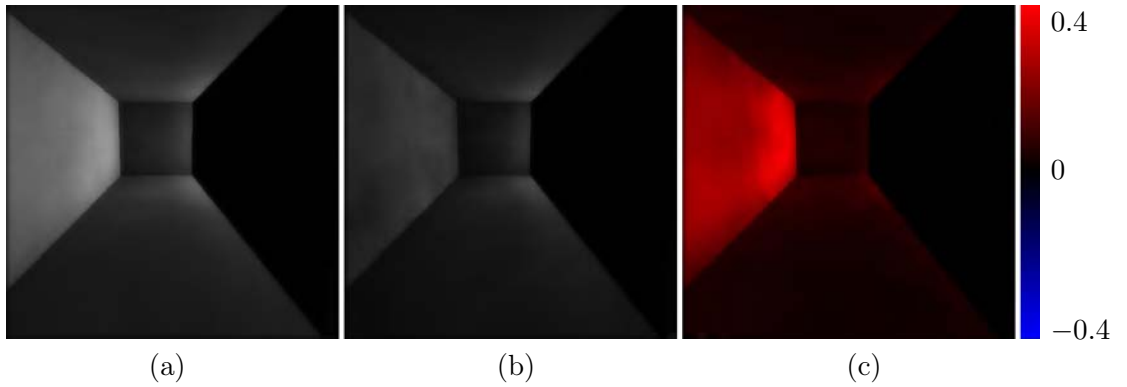


Figure 9.4: The U-shaped scene rendered with the same settings as in the figure 9.3 (b), and the same view with the pull-push phase disabled (a); the difference is significant since holes in the ISM are not filled.

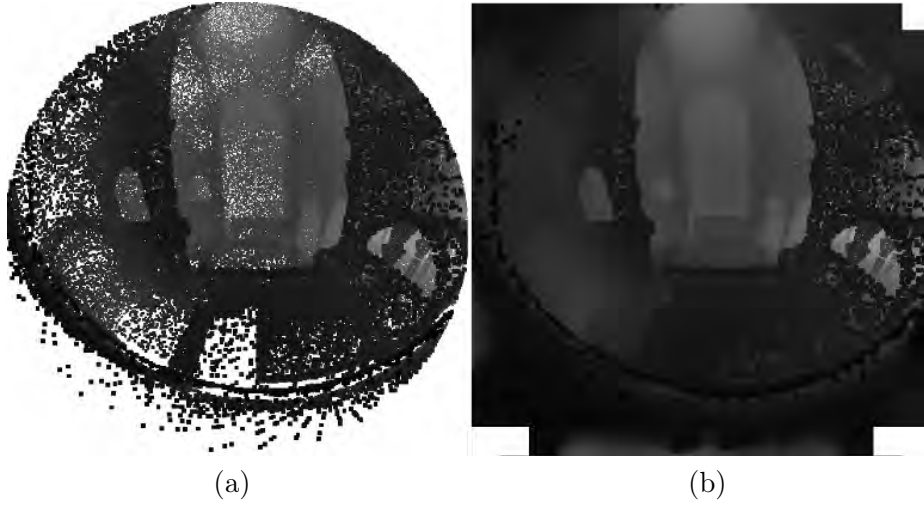


Figure 9.5: One ISM that was created during the Sibenik Cathedral scene rendering process. The image (a) shows the ISM without the pull-push phase, the image (b) shows the same ISM with the pull-push phase enabled. The ISM phase took 3.6 ms without the pull-push phase and 6.8 ms with the pull-push phase.

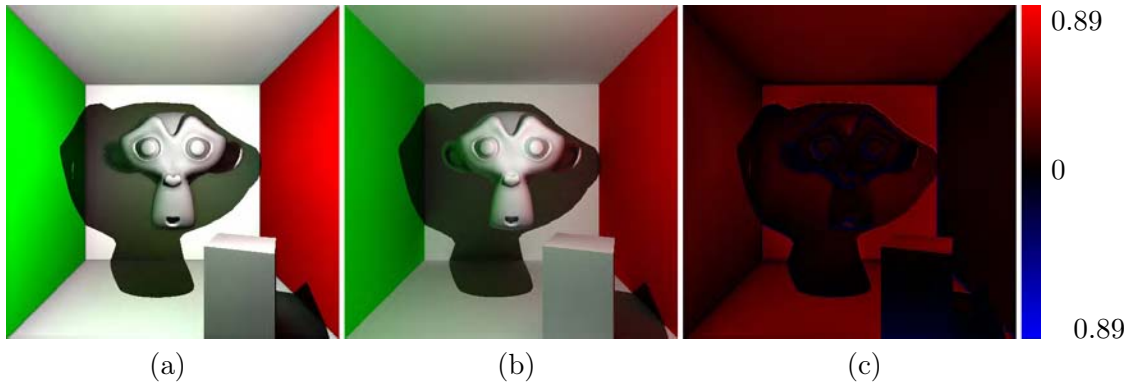


Figure 9.6: The image of the Monkey Box scene rendered using our ISM implementation (a) compared to the reference image, produced by our path-tracer (b); the path-length was set to match the ISM possibilities, so it terminates the tracing after the second bounce.

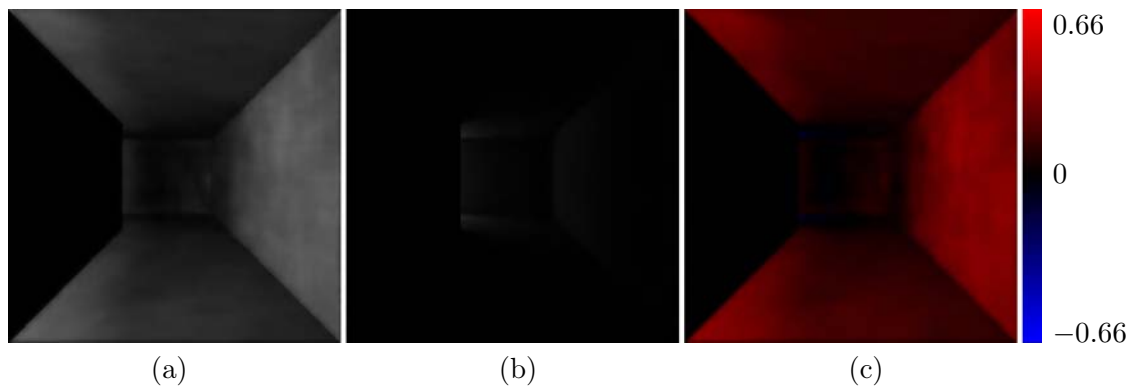


Figure 9.7: The output of our ISM implementation (a) compared to the reference image (b) of the U-shaped scene; the ISM renderer produces much brighter image, as the light "leaks" through the wall.

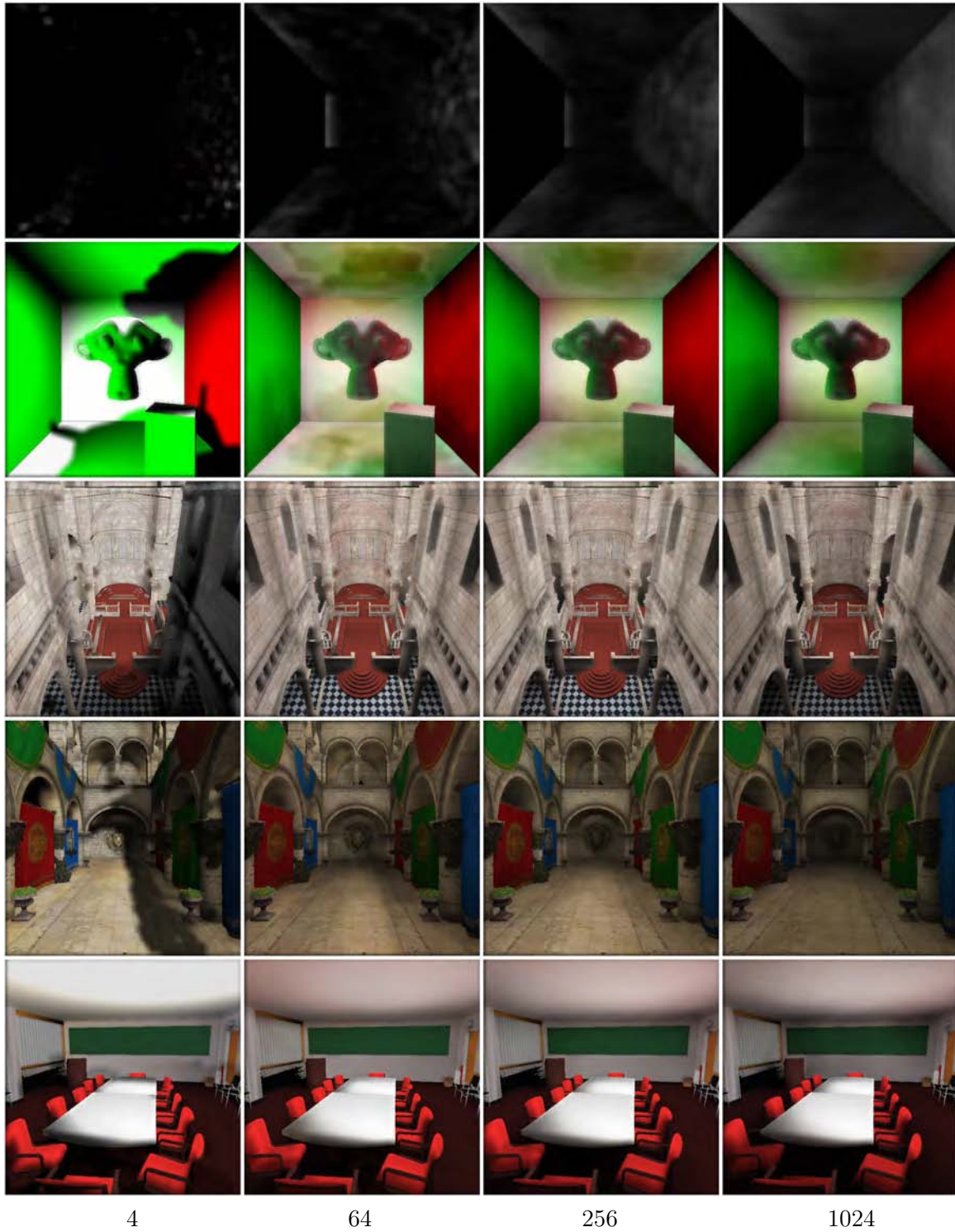


Figure 9.8: The indirect illumination rendered for all tested scenes; the number of VPLs was varied from 4 to 1024. Individual shadows are distinguishable when the VPL count is set too low. Hundreds of VPLs are usually needed to produce an acceptable result.

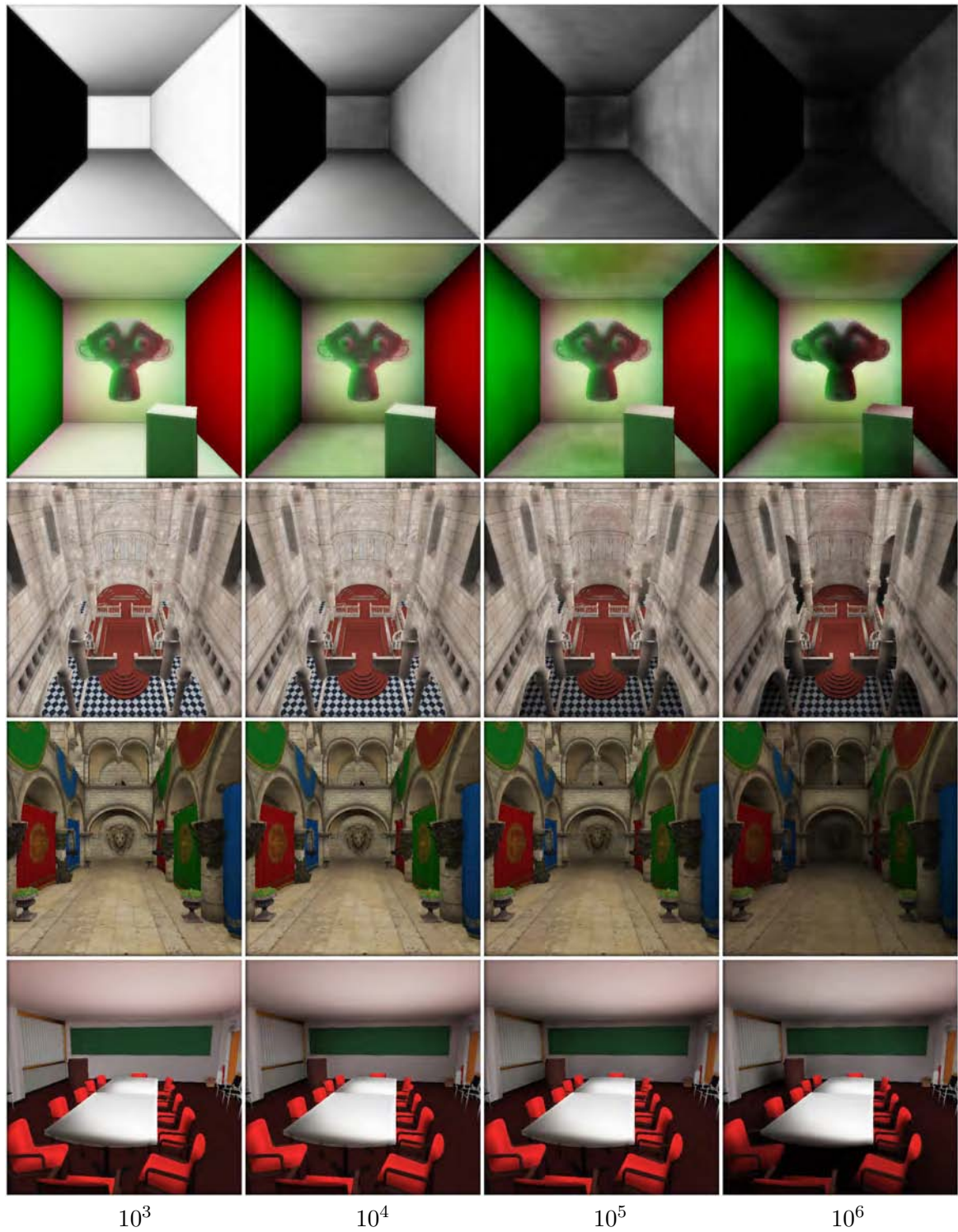


Figure 9.9: The number of point samples varied from 10^3 to 10^6 ; larger scenes require a higher count of point samples, otherwise the indirect shadows are lost. The images show only the indirect illumination.

Chapter 10

Conclusion

Global illumination algorithms have penetrated the field of the real-time applications in recent years a lot. They take the necessary indirect illumination into consideration and produce usually higher quality images, compared to the direct illumination based algorithms. However, physically correct evaluation of the indirect illumination in dynamic scenes still remains a challenging task. Dynamic scenes usually restrict the use of pre-processing, and algorithms designed for such instances typically have to recalculate all the illumination components in each frame. This fact can negatively influence the rendering time.

Most of the reviewed algorithms create a trade-off between the illumination correctness and the rendering time. We have implemented the Imperfect Shadow Maps algorithm [RGK⁺08], which is based on the idea of Instant Radiosity [Kel97]. This algorithm simplifies the visibility checks and creates an imperfect scene representation, which is later splatted into many shadow maps. The ISM algorithm works best with the diffuse surfaces, as it is based on the VPL approach.

10.1 Summary

Our implementation uses the possibilities of a modern GPU (2011) to improve certain parts of the original algorithm. The hardware tessellation unit is used to generate the imperfect point representation on the fly. That increases the performance of the ISM renderer, as it lowers the required memory bandwidth. Also no special treatment for the dynamic objects is required, since the point representation is created directly from the actual geometry setup. The dynamic tessellation also simplifies the refinement of the point representation.

The ISM algorithm suffers from the limited resolution of particular imperfect shadow maps. That means, little or no local lighting events can be captured. On the other hand, screen-space illumination techniques are often appropriate for the evaluation of such events. We combine the ISM renderer with the SSAO technique to enrich the result with a small local lighting features.

Our implementation achieves real-time frame rates on today's hardware setups. Rendering times are typically in terms of tens of milliseconds, depending on a scene. The quality of a produced image does not reach the quality of a reference solution, created by our path-tracer. On the other hand, the ISM renderer is several orders of magnitude faster.

The created application is written as a modular rendering framework. It was tested on several hardware setups and runs at least on Microsoft Windows and Linux platforms. Future improvements should be easy to integrate with the existing code, since we have followed the modern application design patterns, such as the Model-View-Controller architecture.

10.2 Future work

We have found many places where the original algorithm can be improved. The current VPL sampling method estimates the importance of VPLs really coarsely. The challenge is to find an efficient way how to estimate the VPL importance more precisely. The point sample generation has a similar potential for improvements. Considering the temporal coherence, both issues could be solved by an analysis of the previous frames.

Also the SSAO stage has an opportunity for improvements. Recent publications show more possibilities of the local light transfer computed in screen space, such as SSDO and SSDT [RGS09].

The implementation itself could take a benefit from several micro-optimizations. For example, the use of lower precision floating point number formats could create another performance improvements, without significant loss of the image quality.

Bibliography

- [AFO05] Okan Arikan, David A. Forsyth, and James F. O'Brien. Fast and detailed approximate global illumination by irradiance decomposition. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1108–1114, New York, NY, USA, 2005. ACM.
- [Ass09] Assimp Development Team. The assimp library features, 2009. <http://assimp.sourceforge.net/main_features.html>.
- [Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.
- [Bil12] Bill Spitzak. FLTK - Fast Light Toolkit, 2012. <<http://www.fltk.org/>>.
- [BMSW11] Jiří Bittner, Oliver Mattausch, Ari Silvennoinen, and Michael Wimmer. Shadow caster culling for efficient shadow mapping. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 81–88, 2011.
- [BMW09] Jiří Bittner, Oliver Mattausch, and Michael Wimmer. *Game Engine Friendly Occlusion Culling*, chapter Game Engine Friendly Occlusion Culling, pages 637–653. Charles River Media, 1st edition edition, 2009.
- [Boo12] Boost C++ Libraries. Boost Developers, 2012. <<http://www.boost.org/>>.
- [BS09] Louis Bavoil and Miguel Sainz. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, pages 45:1–45:1, New York, NY, USA, 2009. ACM.
- [CMa12] CMake Developers. CMake, 2012. <<http://www.cmake.org/>>.
- [Dev12] DevIL Developers. Devil feature list, 2012. <<http://openil.sourceforge.net/features.php>>.
- [DKH⁺10] Tomáš Davidovic, Jaroslav Křivánek, Miloš Hašan, Philipp Slusallek, and Kavita Bala. Combining global and local virtual lights for detailed glossy illumination. *ACM Trans. Graph.*, 29(6):143:1–143:8, December 2010.
- [DS05] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 203–231, New York, NY, USA, 2005. ACM.

- [DS06] Carsten Dachsbacher and Marc Stamminger. Splatting indirect illumination. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 93–100, New York, NY, USA, 2006. ACM.
- [DSDD07] Carsten Dachsbacher, Marc Stamminger, George Drettakis, and Frédo Durand. Implicit visibility and antiradiance for interactive global illumination. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.
- [DWS⁺88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 21–30, New York, NY, USA, 1988. ACM.
- [Fer05] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.
- [Fre12] FreeImage. Freeimage, 2012. <<http://freeimage.sourceforge.net/>>.
- [GBP06] Gael Guennebaud, Loic Barthe, and Mathias Paulin. Realtime soft shadow mapping by backprojection. In *Eurographics Symposium on Rendering*, pages 227–234, 2006.
- [GCT86] Donald P. Greenberg, Michael F. Cohen, and Kenneth E. Torrance. Radiosity: A method for computing global illumination. *The Visual Computer*, 2:291–297, 1986. 10.1007/BF02020429.
- [Gla93] Andrew S. Glassner. *Graphics gems*, volume 1. Morgan Kaufmann, 1993.
- [GLE12] GLEW Authors. The OpenGL extension wrangler library, 2012. <<http://glew.sourceforge.net/>>.
- [GS10] Iliyan Georgiev and Philipp Slusallek. Simple and Robust Iterative Importance Sampling of Virtual Point Lights. *Proceedings of Eurographics (short papers)*, 2010.
- [GTK12] GTK+ Team. The GTK+ Project, 2012. <<http://www.gtk.org/>>.
- [Hal64] Jonathan H Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, December 1964.
- [HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.
- [Jen01] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.
- [KA06] Janne Kontkanen and Timo Aila. Ambient occlusion for animated characters. In *Proc. Eurographics Symposium on Rendering 2006*, pages 343–348. Eurographics Association, 2006.

- [Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Computer Graphics*, 20(4):143–150, August 1986.
- [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*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.
- [Kel97] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.
- [Khr12a] Khronos Group. GLUT - The OpenGL Utility Toolkit, 2012. <<http://www.opengl.org/resources/libraries/glut/>>.
- [Khr12b] Khronos Group. OpenGL ES - The Standard for Embedded Accelerated 3D Graphics, 2012. <<http://www.khronos.org/opengles/>>.
- [Khr12c] Khronos Group. OpenGL overview, 2012. <<http://www.opengl.org/about/>>.
- [Khr12d] Khronos Group. OpenGL Shading Language, 2012. <<http://www.opengl.org/documentation/glsl/>>.
- [Khr12e] Khronos Group. WebGL - OpenGL ES 2.0 for the Web, 2012. <<http://www.khronos.org/webgl/>>.
- [Laf96] Eric Lafortune. Mathematical models and monte carlo algorithms for physically based rendering. Technical report, 1996.
- [LK11] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 79–88, New York, NY, USA, 2011. ACM.
- [LSK⁺07] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Proc. Eurographics Symposium on Rendering 2007*, pages 277–286. Eurographics Association, 2007.
- [LW93] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS '93)*, pages 145–153, 1993.
- [Mit09] M. Mittring. A bit more deferred-CryEngine 3. In *Triangle Game Conference*, volume 4, 2009. <<http://www.crytek.com/cryengine/presentations>>.
- [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*, pages 101–108, September 2007.
- [Nok11] Nokia Corporation and/or its subsidiaries. Qt online reference documentation, 2011. <<http://doc.qt.nokia.com/>>.

- [NVI09] NVIDIA. Nvidia's next generation CUDA compute architecture: Fermi, 2009. <http://www.nvidia.com/object/fermi_architecture.html>.
- [NVI10] NVIDIA. Nvidia CUDA C programming guide, 2010. <<http://developer.nvidia.com/nvidia-gpu-computing-documentation>>.
- [Ope11] OpenGL wiki authors. History of OpenGL, 2011. <http://www.opengl.org/wiki/History_of_OpenGL>.
- [PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation (The Interactive 3d Technology Series)*. Morgan Kaufmann, August 2004.
- [Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.
- [REH⁺11] Tobias Ritschel, Elmar Eisemann, Inwoo Ha, James D. K. Kim, and Hans-Peter Seidel. Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes. *Computer Graphics Forum*, 30(8):2258–2269, 2011.
- [RGK⁺08] Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008)*, 27(5), 2008.
- [RGKM07] Tobias Ritschel, Thorsten Grosch, Jan Kautz, and Stefan Muller. Interactive illumination with coherent shadow maps. In *Rendering Techniques'07*, pages 61–72, 2007.
- [RGKS08] Tobias Ritschel, Thorsten Grosch, Jan Kautz, and Hans-Peter Seidel. Interactive global illumination based on coherent surface shadow maps. In *Proceedings of graphics interface 2008*, GI '08, pages 185–192, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.
- [RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating Dynamic Global Illumination in Screen Space. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2009.
- [SA11] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification (version 4.2 (core profile), 2011. <<http://www.opengl.org/registry/doc/glspec42.core.20110808.pdf>>.
- [SAG94] Brian Smits, James Arvo, and Donald Greenberg. A clustering algorithm for radiosity in complex environments. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 435–442, New York, NY, USA, 1994. ACM.
- [SIMP06] B. Segovia, J. C. Iehl, R. Mitanchey, and B. Peroche. Bidirectional instant radiosity. In *Proceedings of the 17th Eurographics Workshop on Rendering*, pages 389–398, 2006.

- [SIP06] Benjamin Segovia, Jean-Claude Iehl, and Bernard Peroche. Non-interleaved Deferred Shading of Interleaved Sample Patterns. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware '06*, pages 53–60, September 2006.
- [SSKS06] László Szécsi, László Szirmay-Kalos, and Mateu Sbert. Light animation with precomputed light paths on the gpu. In *Proceedings of Graphics Interface 2006*, GI '06, pages 187–194, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.
- [SSMW09] Daniel Scherzer, Michael Schwärzler, Oliver Mattausch, and Michael Wimmer. Real-time soft shadows using temporal coherence. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, ISVC '09, pages 13–24, Berlin, Heidelberg, 2009. Springer-Verlag.
- [ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.
- [Sto04] William Adams Stokes. Perceptual illumination components: a new approach to efficient, high quality global illumination rendering. *ACM Trans. Graph.*, 23:742–749, 2004.
- [Tsc12] David Tschumperlé. The cimg library, 2012. <<http://cimg.sourceforge.net/>>.
- [WRC88] G.J. Ward, F.M. Rubinstein, and R.D. Clear. A ray tracing solution for diffuse interreflection. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 85–92. ACM, 1988.
- [wxW12] wxWidgets community. wxWidgets - Cross-Platform GUI library, 2012. <<http://www.wxwidgets.org/>>.
- [ZH10] M. Zlatuska and V. Havran. Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms. In *Proceedings of WSCG'2010, communication papers*, pages 69–76, Feb 2010.

Appendix A

List of Abbreviations

AO	Ambient Occlusion
API	Application Programming Interface
BRDF	Bidirectional Reflectance Distribution Function
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CSG	Constructive Solid Geometry
CSM	Coherent Shadow Maps
CSSM	Coherent Surface Shadow Maps
CUDA	Compute Unified Device Architecture
FPS	Frames per Second
G-Buffer	Geometry Buffer
GLSL	OpenGL Shading Language
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
GUI	Graphical User Interface
ISM	Imperfect Shadow Maps
LPV	Light Propagation Volumes
MOC	Meta Object Compiler
MRT	Multiple Render Targets
MVC	Model-View-Controller
RSM	Reflective Shadow Map
SDK	Software Development Kit
SM	Streaming Multiprocessor
SSAO	Screen Space Ambient Occlusion
SSDO	Screen Space Directional Occlusion
SSDT	Screen Space Diffuse Transfer
UI	User Interface
VAISM	View-Adaptive Imperfect Shadow Maps
VFC	Viewing Frustum Culling
VPL	Virtual Point Light

Appendix B

Image Gallery

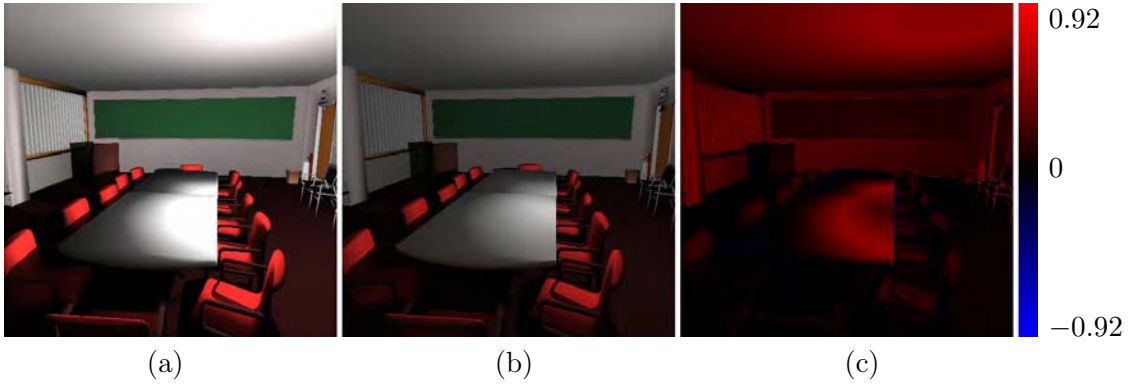


Figure B.1: The Conference Room scene rendered with our ISM renderer (a), our reference path-tracer (b) and the difference image (c); the images were rendered using the direct illumination and the first diffuse bounce of indirect illumination.

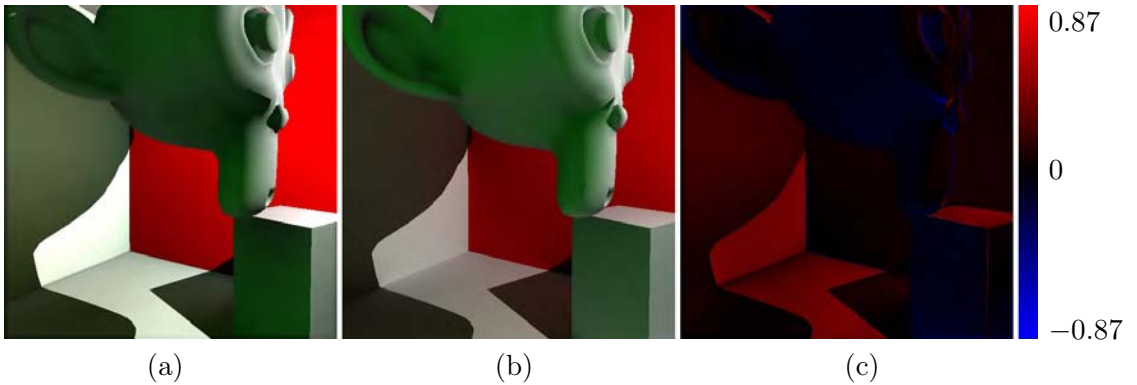


Figure B.2: The Monkey Box scene rendered with our ISM renderer (a), our reference path-tracer (b) and the difference image (c); the images were rendered using the direct illumination and the first diffuse bounce of indirect illumination.

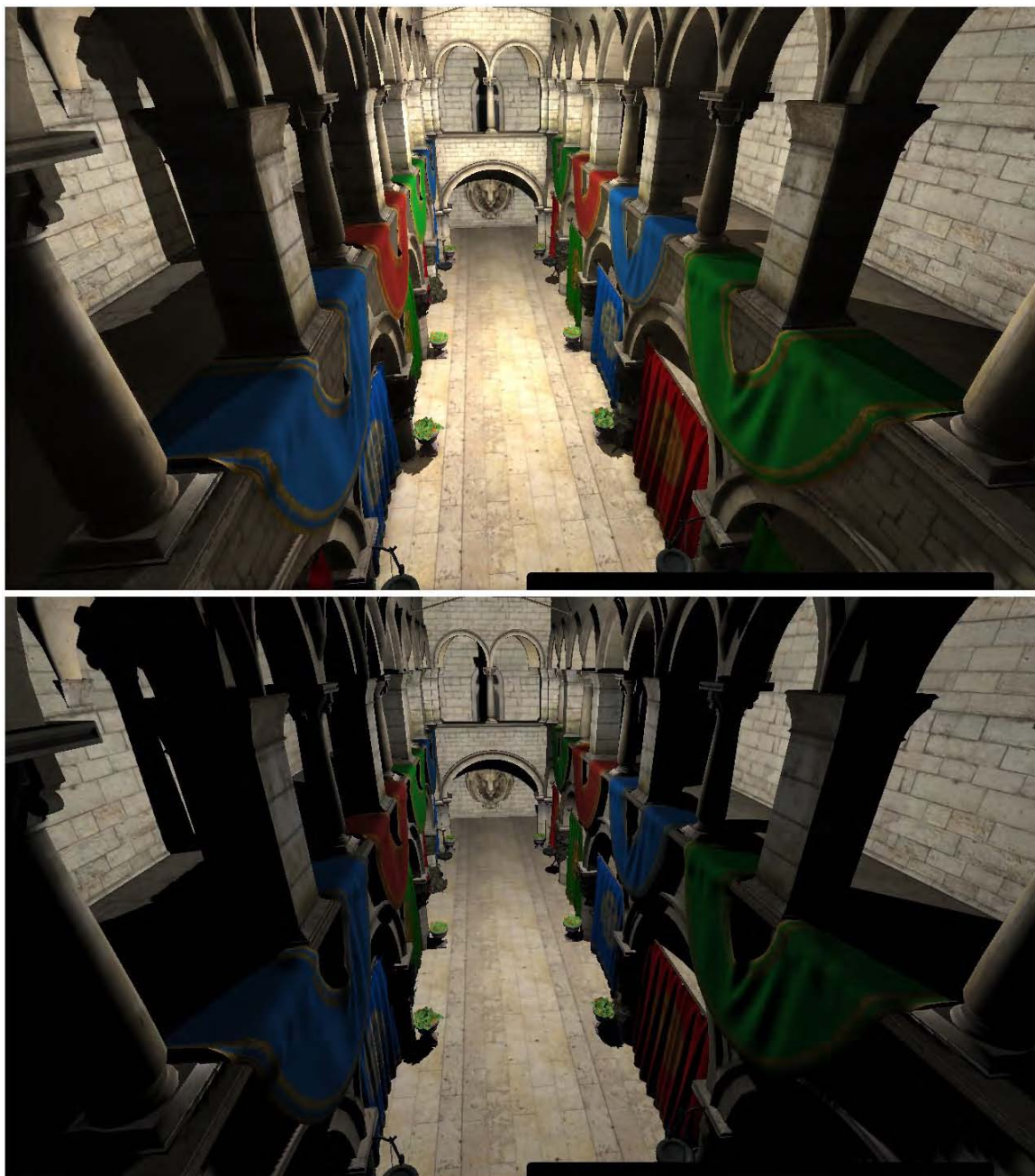


Figure B.3: The Crytek Sponza scene rendered with our ISM algorithm (top) and without the indirect illumination (bottom)

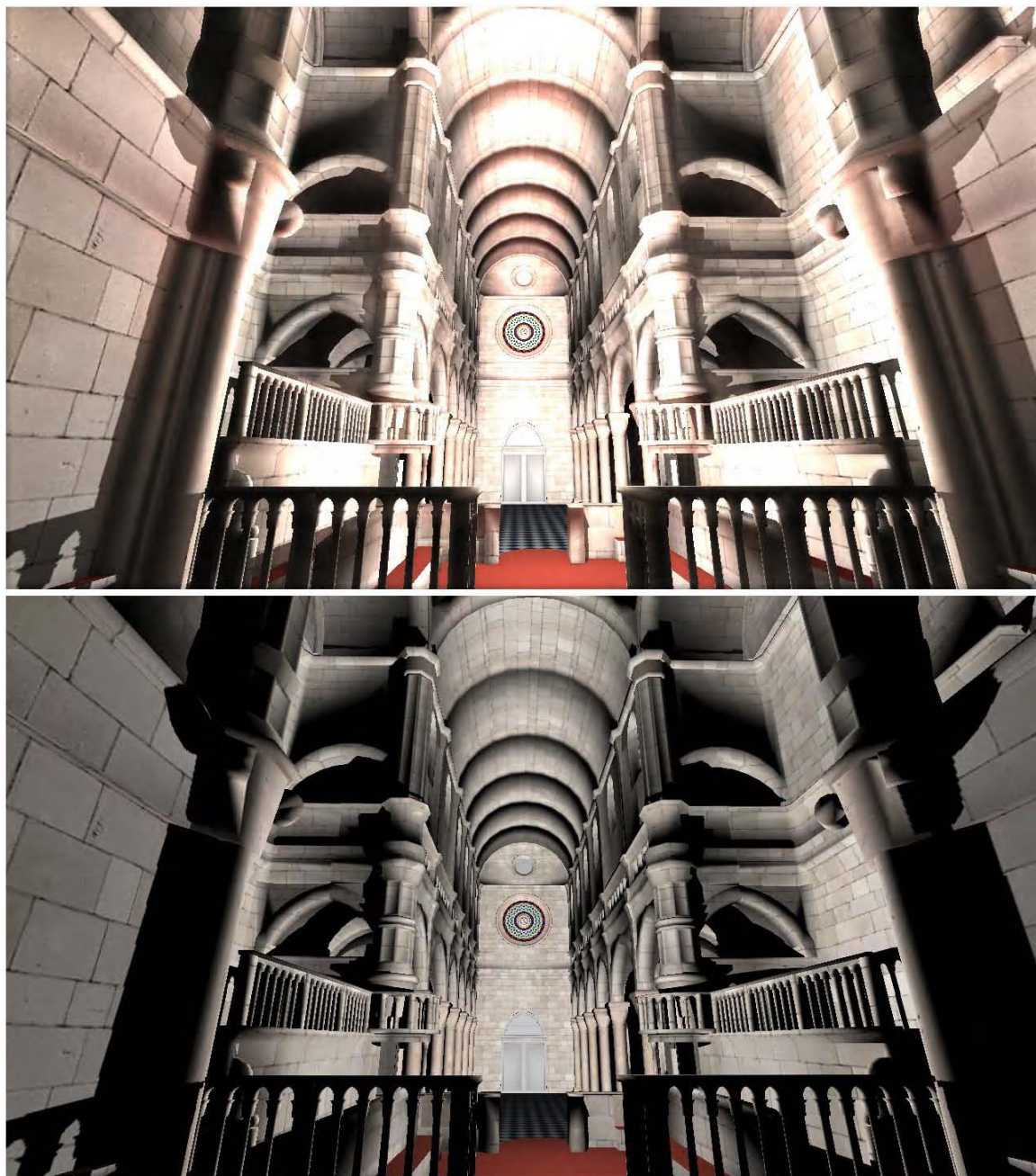


Figure B.4: The Sibenik scene rendered with our ISM algorithm (top) and without the indirect illumination (bottom)



Figure B.5: The Crytek Sponza scene passage lit mostly by the indirect illumination

Appendix C

Installation and User Manual

The application is shipped in a source code form and as a Microsoft Windows executable. All required third-party dependencies are listed in the table [8.2](#).

C.1 Build Instructions

The application should be compilable by all recent versions of the GNU GCC compiler suite. We have tested the 4.4.6 and 4.5.3 versions. On Windows platforms users should use the compiler whipped with Visual Studio 2008 or 2010. The build is managed by CMake. CMake can be used from a command line or with the help of its GUI.

C.1.1 Console Build on Unix-Like Platforms

If you are on a Unix-like platform and you want to build our application from a command line, then open your terminal and issue the following commands:

```
cd /path/to/the/source/code/  
cd build  
cmake ..  
make -j2
```

If any of the required libraries is not found, `cmake` command should fail and output a message about what is missing. The `make -j2` command should build our application. The `-j2` argument is optional and tells the `make` program to build the application in two separate processes. You can tweak this parameter to fit your hardware possibilities, correct setting usually accelerates the build process on multi-core systems.

C.1.2 Using CMake-GUI

Another way is to use the `cmake-gui` application and its graphical interface. A screenshot of this application is shown in the figure [C.1](#). To build our application with the help of the `cmake-gui` application launch the `cmake-gui` application. Use the GUI to set the source directory. The build tree should be located in the build directory. If both directories are

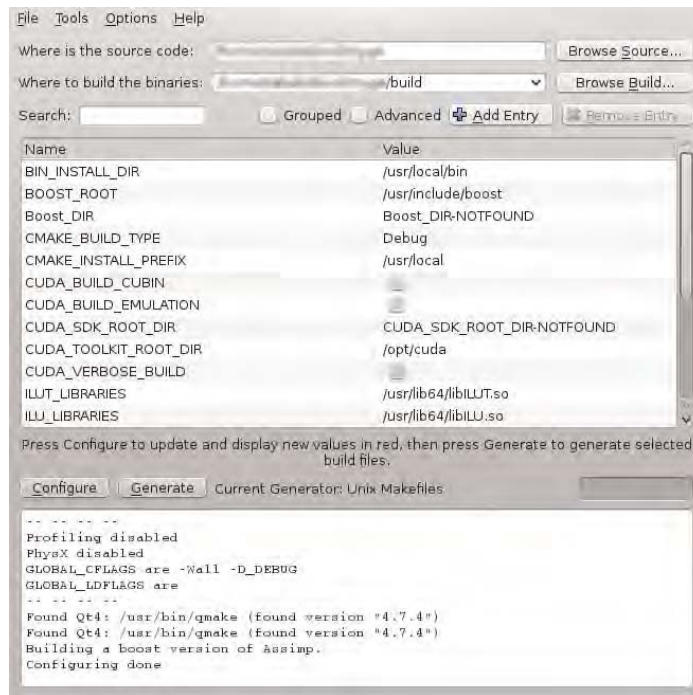


Figure C.1: The cmake-gui application

set correctly, press the **Configure** button. A list of variables should appear in the central widget. The log similar to the console command output appears in the lower widget. If any of the third-party dependencies is not found, appropriate error message should be shown. If the configuration went ok, press the **Generate** button. CMake creates Unix Makefiles on Unix systems by default. Read the section [C.1.3](#) for the Windows build instructions. Open your favourite terminal and issue following commands:

```
cd /where/have/you/set/the/build/directory/
make -j2
```

The application should compile in the same way as in with the command line `cmake` approach. If your build was successful, the `qtgui` executable should appear in your build directory.

C.1.3 Build on Windows Platforms

After the installation of all required dependencies you should use the `cmake-gui` application to generate your Visual Studio project files. Start the `cmake-gui` application and follow the instructions written in the section [C.1.2](#). Use one of the supported Visual Studio generators and create the project files. Open the `.sln` file in Visual Studio and build the `qtgui` application. Please note that all third-party `dll` files have to be reachable from your `PATH` system variable.

C.2 Usage

If you want to run the shipped executable, go to `win32release` directory and execute the `qtgui` application. Please ensure that the working directory is set to the project directory if you want to execute the application from the Visual Studio IDE. The application tries to access the data opening the `../data` path on Windows platforms. Execute the `qtgui` application from the directory which contains the `data` sub-directory if you work on any Unix-like platform.

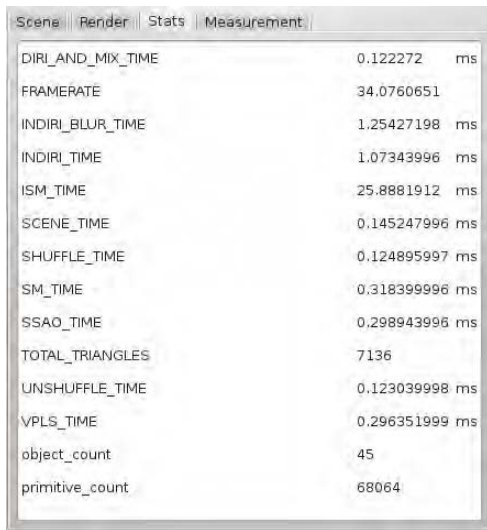
The application creates two windows, one showing the rendered scene and one tool window. The 3D viewport reads the mouse and keyboard input which can be used to control the camera and light movement. Drag a mouse to rotate the camera, use the `W`, `S`, `A`, `D` keys to move the camera. Use the `I`, `K`, `J`, `L` keys to move the light. The tool window contains four tabs and can be used to change several render settings or to select a scene to be displayed. All tabs are shown in the figure [C.2](#). The Stats tab shows the breakdown of the current scene rendering time. The Measurement tab can be used to automatize the measurement process.



(a) The Scene tab



(b) The Render tab



(c) The Stats tab



(d) The Measurement tab

Figure C.2: Four tabs of the tool window of our application

Appendix D

Contents of Attached CD

baraktom-thesis/	- the project root directory
--build/	- directory for the build tree
--data/	- shaders and testing scenes
--ism/	- shaders for the ISM renderer
--ogl3rend/	- common shaders
'--scenes/	- testing scenes
--uscn/	- the U-shaped scene
'--monkey_box/	- the Monkey Box scene
--dep/	- enclosed third-party libraries
--assimp/	- Assimp sources
--lib-linux-x86/	- Linux 32 libraries
--lib-linux-x86_64/	- Linux 64 libraries
--include/	- third-party includes
'--lib-windows-x86/	- Win 32 libraries
--doc/html/	- generated code documentation
--src/	- source codes
--apps/	- source codes for applications
--qtgui/	- the main application with GUI
'--imdiff/	- tool for generating image diffs
'--libs/core/	- the most important code
--thesis/	- this thesis sources and pdf
--win32release/	- application compiled for Win 32
'--README	- build instructions