



**CZECH TECHNICAL  
UNIVERSITY  
IN PRAGUE**

**F3**

**Faculty of Electrical Engineering  
Department of Computer Graphics and Interaction**

**Master's Thesis**

# **Real-time Global Illumination using Irradiance Probes**

**Šimon Sedláček**  
Open Informatics

**May 2019**

**Supervisor: doc. Ing. Jiří Bittner, Ph.D.**



## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Sedláček** Jméno: **Šimon** Osobní číslo: **434870**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**  
Studijní program: **Otevřená informatika**  
Studijní obor: **Počítačová grafika**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Výpočet globálního osvětlení v reálném čase pomocí sond intenzity ozáření**

Název diplomové práce anglicky:

**Real-Time Global Illumination using Irradiance Probes**

Pokyny pro vypracování:

Review existing methods for real-time global illumination. Focus on methods that allow computing global illumination even on low-end graphics hardware and support at least partially dynamic scenes. Implement a recent global illumination method using precomputed local reconstruction from sparse radiance probes [1]. Conduct a series of tests of the method on at least two different scenes and evaluate its precomputation times and running times. Evaluate the rendering quality of the methods by comparing with an offline global illumination solution or using high-quality settings of the implemented method with many radiance probes. Identify the strengths and weaknesses of the method. Try to invent an extension of the implemented method which would allow using non-static objects and/or extensions aimed for friendly real-life usage of this algorithm.

Seznam doporučené literatury:

- [1] Ari Silvennoinen, Jaakko Lehtinen. Real-time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes. ACM Transactions on Graphics, Vol. 36, No. 6, Article 230, 2017.
- [2] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. ACM Transactions on Graphics, Vol. 36, No. 6, Article 230, 2017.
- [3] Jaroslav Křivánek, Kadi Bouatouch, Sumanta Pattanaik, and Jiří Žára. Making radiance and irradiance caching practical: Adaptive caching and neighbor clamping. In Rendering Techniques 2006 (Proc. of Eurographics Symposium on Rendering), pages 127–138, 2006
- [4] Tobias Ritschel Carsten Dachsbacher Thorsten Grosch Jan Kautz. The state of the art in interactive global illumination. In Computer Graphics Forum (Vol. 31, No. 1, pp. 160-188), 2012.

Jméno a pracoviště vedoucí(ho) diplomové práce:

**doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.02.2019**

Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce: **20.09.2020**

doc. Ing. Jiří Bittner, Ph.D.  
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.  
podpis děkana(ky)

### III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.  
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

\_\_\_\_\_  
Datum převzetí zadání

\_\_\_\_\_  
Podpis studenta

## Acknowledgement / Declaration

I would like to thank my supervisor doc. Ing. Jiří Bittner, Ph.D. as he was supportive during my work on this thesis and encouraged me to continue in this field of study beyond the scope of this thesis. I would also like to thank my family and my lovely Baruška for being so supportive during all those years I have spent studying. Who would have thought that their support would result in real-time global illumination?

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 15. 5. 2019

.....

## Abstrakt / Abstract

V této práci prezentuji svá dvě vylepšení k nedávno zveřejněné metodě pro výpočet globálního osvětlení založené na ukládání iradiance v řídké množině. Mé první rozšíření umožňuje výpočet globálního osvětlení i pro dynamické objekty. Toto rozšíření podporuje velké množství stínovacích technik (normálové mapování, spekulární odrazy). Mé druhé rozšíření používá dynamické úrovně detailu pro globální osvětlení. Je schopno výrazně zvýšit výkon algoritmu bez velkých chyb ve vykreslených snímcích.

**Klíčová slova:** globální osvětlení v reálném čase, předpočítaný transport radiance, přenos světla, sférické harmonické, dynamická úroveň detailu, zlepšení časové náročnosti

**Překlad titulu:** Výpočet globálního osvětlení v reálném čase pomocí sond intenzity ozáření

I present two extensions to a recently introduced method for real-time global illumination based on sparse irradiance caching. The first extension allows the computation of global illumination shading for dynamic objects with the support of various shading techniques (normal mapping, specular reflections). The second extension uses a dynamic level of detail for global illumination, which is able to significantly improve performance without noticeable errors of a rendered frame.

**Keywords:** real-time global illumination, precomputed radiance transfer, light transport, spherical harmonics, dynamic level of detail, performance improvement

# Contents /

<b>1 Introduction</b> .....	1
<b>2 Related Work</b> .....	2
2.1 Overview of Real-Time Global Illumination Methods.....	2
2.2 Precomputed Radiance Transfer .....	4
<b>3 Real-Time Global Illumination by Precomputed Local Recon- struction from Sparse Radi- ance Probes</b> .....	5
3.1 Method Overview .....	5
3.1.1 Quick introduction.....	5
3.1.2 Detailed introduction .....	7
3.2 Precomputation Phase.....	12
3.2.1 Irradiance Receivers Placement .....	12
3.2.2 Probe Placement & Radius Calculation .....	16
3.2.3 Probe Ray-Casting & Receiver Coefficients.....	18
3.2.4 Receiver Clustering .....	21
3.2.5 Computational Tex- tures and Output Data File .....	25
3.3 Rendering in Real-Time .....	30
3.3.1 Main Rendering Loop ...	30
3.4 Asymptotic Time and Mem- ory Complexity .....	34
3.4.1 Precomputation Algo- rithm Complexity .....	34
3.4.2 Real-time Algorithm Complexity .....	35
<b>4 Method Extensions</b> .....	36
4.1 Shading Dynamic Objects Using Voxel Irradiance Space .	37
4.1.1 Irradiance Voxel Space Using Regular Grid .....	38
4.1.2 Computation of Irradi- ance from Voxel Grid ....	40
4.2 Dynamic Level of Detail.....	41
4.2.1 Probe Computation with LOD .....	41
4.2.2 Cluster Basis Compu- tation with LOD .....	44
4.2.3 Irradiance Receiver Computation with LOD .	45
4.2.4 Usage of LOD Irradi- ance Textures in Frag- ments .....	47
<b>5 Results</b> .....	48
5.1 Real-Time Global Illumina- tion by Precomputed Local Reconstruction from Sparse Radiance Probes .....	49
5.1.1 Testing Number of Probes .....	49
5.1.2 Testing Number of Re- ceivers .....	50
5.1.3 Testing $\alpha_{ij}$ Coefficients Setup .....	52
5.1.4 Incorrect Irradiance at Contact Points of Meshes .....	54
5.2 Extensions.....	56
5.2.1 Global Illumination Shading for Dynamic Objects .....	56
5.2.2 Dynamic Level of Detail .	61
<b>6 Conclusion</b> .....	71
6.1 Future Work .....	71
<b>References</b> .....	72
<b>A The Application Guide</b> .....	75
A.1 Scene Configuration File.....	75
A.2 Application GUI .....	77
<b>B Used Libraries and Extern Packages</b> .....	78
<b>C Compilation of Provided Appli- cations</b> .....	79
<b>D The Contents of The Enclosed DVD</b> .....	80

## Tables / Figures

<b>5.1.</b> Setups for testing number of probes in the scene.....	49	<b>1.1.</b> The proposed method example ..	1
<b>5.2.</b> Setups for testing number of receivers in the scene.....	51	<b>2.1.</b> The proposed method by Silvennoinen and Lehtinen .....	4
<b>5.3.</b> Table of density setup for irradiance voxel grid testing ...	57	<b>3.1.</b> Simple visualization of rendering cycle. ....	6
<b>5.4.</b> Open Hall testing LOD setups .	61	<b>3.2.</b> Simple visualization of rendering cycle, second iteration. ...	6
<b>5.5.</b> Computation times for renders in figure of Open Hall, camera 1 .....	62	<b>3.3.</b> Visualization of SH degrees. ....	8
<b>5.6.</b> Computation times for renders of Open Hall, camera 2 .....	63	<b>3.4.</b> Example of SH basis functions. ..	8
<b>5.7.</b> Computation times for renders of Open Hall, camera 3 .....	64	<b>3.5.</b> Visualization of probe's re-light rays. ....	9
<b>5.8.</b> Computation times for renders of Cornell Box scene. ....	65	<b>3.6.</b> Visualization of visibility match between probes and a receiver. ....	10
<b>5.9.</b> Computation times, for renders of Large Cornell Box scene. ....	66	<b>3.7.</b> Rule 1 of lightmap UV coordinates calculation .....	13
<b>5.10.</b> Table of distances setup for LOD testing .....	67	<b>3.8.</b> Rule 2 of lightmap UV coordinates calculation .....	13
<b>5.11.</b> Table of SH degree setup for LOD testing .....	68	<b>3.9.</b> Unused empty space vis. in UV map .....	14
		<b>3.10.</b> Fill of unused empty space vis. in UV map .....	14
		<b>3.11.</b> UV coordinates without border .....	15
		<b>3.12.</b> UV coordinates with border ...	15
		<b>3.13.</b> Voxel normal consistency check. ....	16
		<b>3.14.</b> Probe placement .....	17
		<b>3.15.</b> Probe Ray-casting.....	18
		<b>3.16.</b> Receiver, support of probes....	19
		<b>3.17.</b> Receiver, casting rays .....	19
		<b>3.18.</b> Clustering visualization, without error base splitting....	21
		<b>3.19.</b> Clustering visualization, with error base splitting .....	22
		<b>3.20.</b> Clustering comparison .....	22
		<b>3.21.</b> Clustering, initial transport matrix .....	23
		<b>3.22.</b> Probe Ray Intersection texture .....	25
		<b>3.23.</b> Spherical Harmonics texture...	25
		<b>3.24.</b> Cluster Matrix texture.....	26
		<b>3.25.</b> Cluster Size texture .....	26
		<b>3.26.</b> Cluster Probe Index texture...	27
		<b>3.27.</b> Receiver Map texture .....	27
		<b>3.28.</b> Receiver Transport texture ....	28



<b>3.29.</b>	Receiver Material Map texture .....	28
<b>3.30.</b>	Main rendering loop .....	30
<b>3.31.</b>	Cluster basis vectors: matrix multiplication .....	31
<b>3.32.</b>	Irradiance fetch-back examples .....	33
<b>4.1.</b>	Visualization of spatial receiver .....	37
<b>4.2.</b>	Visualization of active spatial receivers .....	39
<b>4.3.</b>	Reconstruction of irradiance for dynamic objects .....	40
<b>4.4.</b>	LOD probe distance.....	41
<b>4.5.</b>	LOD probe's relight rays .....	42
<b>4.6.</b>	LOD receivers supported by probe .....	42
<b>4.7.</b>	LOD receivers supported by probe, shifted by radius.....	43
<b>4.8.</b>	LOD Camera View Dependant Zones.....	43
<b>4.9.</b>	LOD cluster matrix .....	44
<b>4.10.</b>	LOD receiver write thought irradiance texture LOD .....	45
<b>4.11.</b>	LOD receiver in different LOD than its cluster .....	46
<b>4.12.</b>	Fragments take irradiance without smoothing between LOD .....	47
<b>4.13.</b>	Fragments take irradiance with smoothing between LOD .	47
<b>5.1.</b>	Screenshot from the implemented application. ....	48
<b>5.2.</b>	Testing renders for different number of probes.....	49
<b>5.3.</b>	Testing scenes for a different number of probes with visualized probes position.....	49
<b>5.4.</b>	Testing scenes for a different number of probes, single setup .	50
<b>5.5.</b>	Testing scenes for a different number of probes, multiple setups .....	50
<b>5.6.</b>	Testing renders for a different number of receivers .....	50

<b>5.7.</b>	Testing renders for different number of receivers, single setup .....	51
<b>5.8.</b>	Testing renders for a different number of receivers, dependency graph .....	51
<b>5.9.</b>	Receivers, no hitpoint example.....	52
<b>5.10.</b>	Testing weighted function for ray hitpoints, visualization of the problem .....	53
<b>5.11.</b>	Testing weighted function for ray hitpoints, renders .....	53
<b>5.12.</b>	Incorrect irradiance at contact points of meshes, example of a bright aura .....	54
<b>5.13.</b>	Incorrect irradiance at contact points of meshes, how higher sampling dims the aura.....	54
<b>5.14.</b>	Incorrect irradiance at contact points of meshes, shifting receivers position.....	55
<b>5.15.</b>	Comparison of render with shifted receivers position and the original render without shifting.....	55
<b>5.16.</b>	Dynamic irradiance voxel calculation renders .....	56
<b>5.17.</b>	Dynamic irradiance voxel calculation, an SSIM difference .....	56
<b>5.18.</b>	Irradiance voxel grid with different density setups .....	57
<b>5.19.</b>	Irradiance voxel grid with different density setups, active voxels visualization.....	57
<b>5.20.</b>	Dynamic object shading results: Layers of shading.....	58
<b>5.21.</b>	Shadow casted by a dynamic object .....	59
<b>5.22.</b>	Dynamic object shading results: Static light sources .....	59
<b>5.23.</b>	Normal mapping for static scene. Example.....	60

<b>5.24.</b>	Rendered scene Open Hall, camera 1 .....	62
<b>5.25.</b>	Rendered scene Open Hall, camera 2 .....	63
<b>5.26.</b>	Rendered scene Open Hall, camera 3 .....	64
<b>5.27.</b>	Rendered scene Cornell Box with different lightmap size for different LOD.....	65
<b>5.28.</b>	Rendered scene Large Cor- nell Box with different SH degree for LOD 0 only.....	66
<b>5.29.</b>	Different LOD distance se- tups. ....	67
<b>5.30.</b>	Different SH degree setups. ....	68
<b>5.31.</b>	Fragments take irradiance without smoothing between LOD .....	69
<b>5.32.</b>	Camera View Dependant Zones in LOD .....	70
<b>5.33.</b>	Testing lightmap border ex- tension pixels in LOD.....	70
<b>A.1.</b>	Scene configuration file com- mands.....	76



# Chapter 1

## Introduction

Calculation of correct light reflectance through the scene in computer graphics is one of the most impactful effects there is. Correct light transport is often the effect, which blurs a line between computer graphics and reality the most. As impactful this effect is, it is also costly.

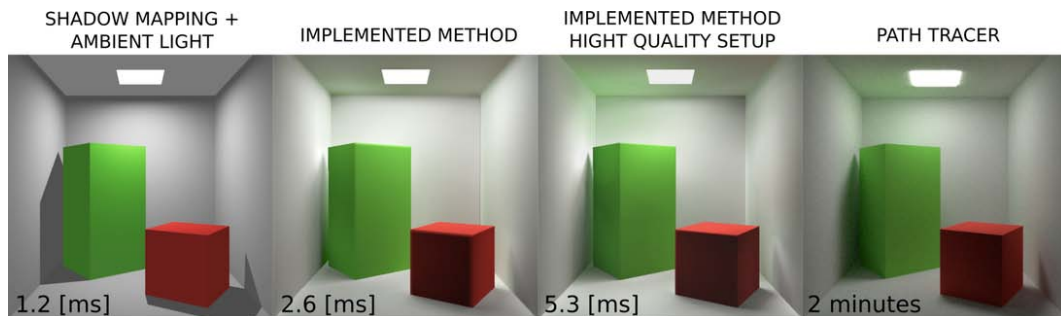
Due to the speed and number of photons, it is almost impossible to compute correctly. There are ways how to approximate light reflectance for offline rendering, for example, path tracing [1]. These methods provide incredible results, but rendering time is adequate to the visual effect. Furthermore, methods like path tracing are prone to noisy outputs due to an insufficient amount of sampling rays. An example is shown in Figure 1.1 right image.

To compute such an enormous task of light reflectance in real-time, even more problems show up. A standard way how to solve the problem today is to calculate only single light bounce, from the light source to a first hit point. The rest of the light reflectance is then approximated with ambient light. This solution is fast but lacks the visual quality as no color bleeding or soft shadows are possible. An example, using shadow mapping [2] and ambient light, is shown in Figure 1.1 left image.

As it is costly to compute the traveling path for every photon in real-time, there was an idea to precompute these paths and reuse them in real-time rendering. Class of algorithms which precompute lightpaths and reuse them in real-time is called Precomputed Radiance Transfer [3].

For a starting point of my work, I chose newly released algorithm from Precomputed Radiance Transfer class called Real-time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes [4] by Silvennoinen and Lehtinen. This algorithm allows to precompute light paths and due to re-rendering of computed light paths, it is able to approximate global illumination in real-time for dynamic light-sources. An example output of this algorithm is shown in Figure 1.1 middle images. As for every method, there are some drawbacks. This method does not support dynamic objects and effects like a normal mapping or specular reflections are costly.

My goal with this algorithm was to create and implement extensions to push it towards real-life usage in commercial rendering engines.



**Figure 1.1.** Rendering of Cornell box using different rendering techniques, rendering times are noted in the bottom left corner.

## Chapter 2

### Related Work

Because the implemented algorithm [4] belongs to the class of algorithms called Pre-computed Radiance Transfer, I separated this chapter into two sections. The first section describes general approaches on how global illumination in real-time has been approximated. The second section focuses solely on the Precomputed Radiance Transfer methods.

### 2.1 Overview of Real-Time Global Illumination Methods

To achieve global illumination effect for low-frequency spatially-varying Bidirectional Reflectance Distribution Function (BRDF) for completely static scenes (only the camera can move), we can use any type of offline rendering method capable of global illumination. An example can be the path tracing algorithm [1]. Light is computed and stored in textures, which are then used during rendering. This process is often called light baking [5].

Even when partial interactivity is needed in real-time global illumination, the problem becomes much more difficult to solve. Based on the paper of Ritschel et al. [6], real-time global illumination methods can be separated into several categories.

One of the first approaches discretized the scene and use only patches (discrete samples) to compute light interaction between objects. This approach is called Finite Elements (sometimes called Radiosity) [7]. Finite Elements method is suitable for diffuse surfaces as patches only calculate patch-to-patch light interaction omitting the viewing position. The number of patches defines the quality of light reconstruction. However, as we need to compute the interaction of every patch with every other patch, this approach does not scale well (quadratic growth). Nichols et al. [8] introduced hierarchical ordering of patches for better scalability of the Finite Elements method.

The second available approach uses ray-tracing algorithms. Ray-tracing algorithms provide a great variety of supported light effects: diffuse surfaces, reflections, refractions or subsurface scattering. However, these algorithms have often noisy outputs, due to insufficient sampling of rays. Usage of ray-tracing algorithms for real-time light reflectance is becoming more available as recently introduced GPUs support HW Bounding Volume Hierarchies (BVH) traversal using dedicated ray-tracing cores.

Photon Mapping [9] algorithm uses the idea of shooting a dense set of photons from light-source which are then stored in a map. During rendering, this map is used to estimate irradiance in certain fragment with respect to the density of photons in the nearest area. Photon Mapping can be used to simulate even high-frequency spatially-varying BRDF (as caustics). The bottleneck of this algorithm is the photon density estimation, as every fragment has to search nearest photons in the photon map.

A similar approach to Photon Mapping is Instant Radiosity method [10]. In Instant Radiosity, photons are not used for density estimation instead they are used as virtual point lights (VPLs). As the name itself suggests, VPLs are then used as light-sources to illuminate fragments of the scene. Using VPLs, Instant Radiosity avoids the searching in the map during rendering. Nevertheless, there is still iteration over all VPLs. Walter et al. [11] introduced Light Cuts method, which builds hierarchical tree structure over all virtual point lights, enabling sub-linear light gathering in fragments.

More general approach with the usage of VPL idea is called Point-based Global Illumination [12]. Here, sample points are not only shot from light-sources but cover the whole scene. A similar hierarchical structure is then used to gather light in fragments.

Notice how all methods above were neglecting the light path. They only measured light at the surface. Because of this, these approaches are not suitable for rendering global illumination in media with different density or viscosity. To capture light traveling through the media, Discrete ordinate methods [13] discretize the space using a voxel grid. Light traveling through the volume is then computed as voxel interaction with all neighboring voxels. This approach yields slower convergence of the system. Kaplanyan and Dachsbacher [14] improved convergence speed by usage of recursive grids together with reflective shadow maps [15].

## 2.2 Precomputed Radiance Transfer

As it was written in the paper of Ritschel et al. [6], many simplifications can be made if all objects positions are fixed. It is possible to precompute light paths and reuse them with precomputed visibility in real-time. Class of algorithms using this idea is called Precomputed Radiance Transfer (PRT). For example, Sloan et al. [16] captured how the surface of a geometry reacts to an incoming radiance. They stored this information on surface points and were able to reconstruct a shading with self-shadowing not only for polygon meshes but also for volumetric geometry such as clouds and fog.

For storage of light paths and information about light reflectance in many PRT methods (including above-mentioned work of Sloan et al. [16]), Spherical Harmonics [17] are often used. Green [18] describes the basic usage of Spherical Harmonics in computer graphics.

Spherical Harmonics are efficient on memory and reconstruction. Though, they are suitable mainly for low-frequency spatially-varying BRDF, such as diffuse surfaces. As one of the first tries to avoid this restriction Liu et al. [19] used non-linear wavelets to capture any-frequency signal. This allowed reconstructing even specular reflections. However, non-linear wavelets are more demanding on both memory and computation time. Which is also mentioned in the paper of Ritschel et al. [6].

Due to local coherence of light transport in space, the signal can be compressed using clustering [20], Principal Component Analysis (PCA) [21] and other methods.

PRT methods do not support dynamic objects due to their precomputed light transfer. To give at least approximation of lighting for dynamic objects Greger et al. [22] stored a grid of spatial irradiance measured in many ways around the measuring point to reconstruct lighting at any point in space. Similar approaches [23] were presented to store ambient occlusion [24] or to transfer entire radiance [25] for reconstruction of glossy materials. Jendersie et al. [26] used the idea of spatial irradiance caching [22] to propagate incoming light to surface samples, from which they interpolated resulting irradiance.

To support fully dynamic lights, Silvennoinen and Lehtinen [4] decomposed the light transport to two major stages. They separated a light gathering (using radiance probes) and a light distribution (using radiance receivers). Furthermore, with the usage of Singular Value Decomposition (SVD), the truncation of transport matrices lead to memory and time efficient algorithm for real-time global illumination. However, this method does not support dynamic objects. Furthermore, effects like normal mapping or specular reflections are possible only if radiance is transported. As it was already noted by authors, transport of radiance is noticeable more demanding on both memory and computation time due to the higher dimension of transport in every stage. Example of this method is shown in Figure 2.1.



**Figure 2.1.** The proposed method by Silvennoinen and Lehtinen [4].



## Chapter 3

# Real-Time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes

Before I explain the implementation details of the proposed method [4], I will describe the method in a more general way for better understanding.

### 3.1 Method Overview

To explain the most general concept of the light propagation, I separated explanation into two sections. Section 3.1.1 provides a general idea without any mathematical equations, whereas section 3.1.2 provides a detailed introduction into the mathematics behind this algorithm.

#### 3.1.1 Quick introduction

To calculate global illumination in real-time, this method uses factorized radiance transport and a sparse set of radiance probes. The method uses two main objects where irradiance is transported:

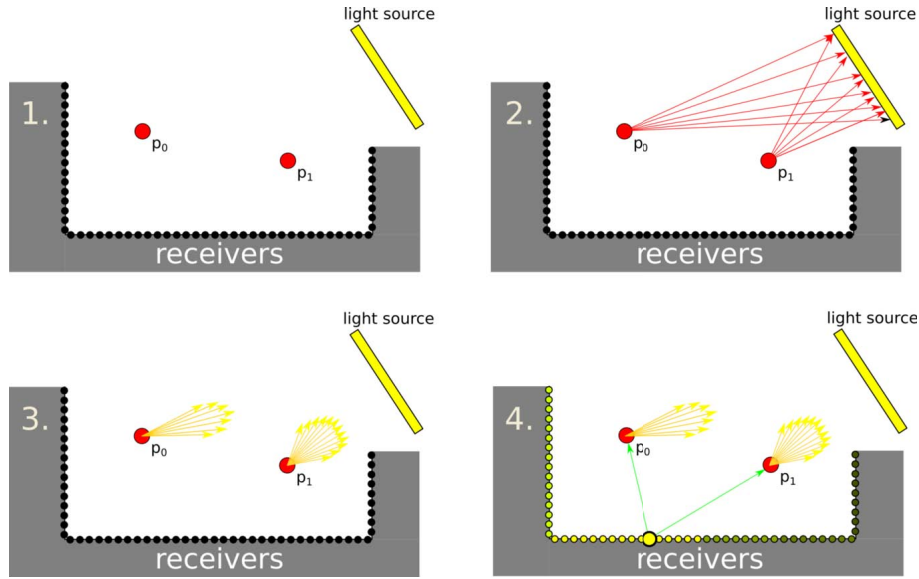
**1) Probes**

**2) Receivers**

A probe is a point in space in which incoming radiance is measured from many directions (ideally it would be from all directions). The probe stores this measured radiance, which is then used by receivers. A set of probes is sparse (the usual density is around 0.02 probes per  $m^3$ ).

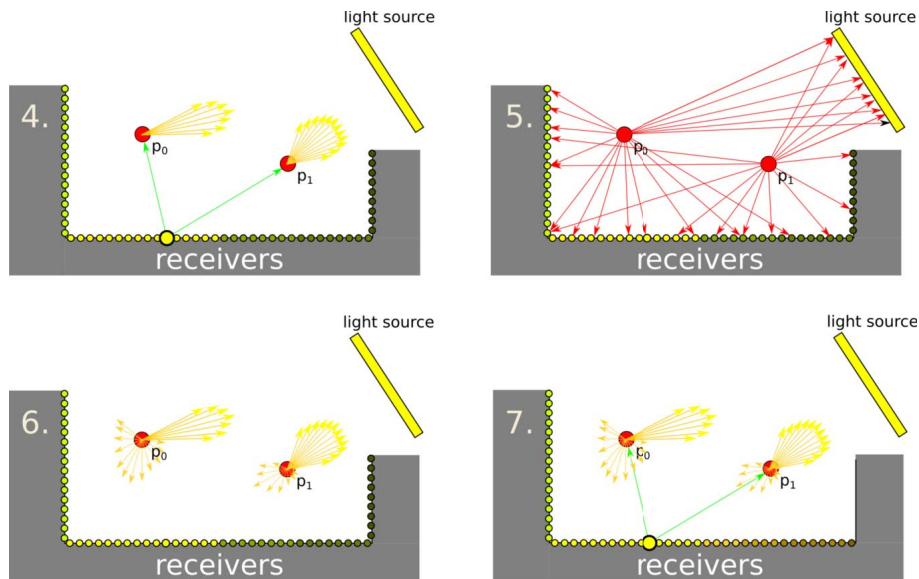
A receiver is a point on the surface of the scene. The receiver uses surrounding probes to gather irradiance. For each probe, the receiver fetches the radiance from a correct direction and accumulate it. A set of receivers is dense (the usual density is around 80 receivers per  $m^3$ ). In the implementation, receivers are texels of a lightmap, which is then used when meshes of the scene are rendered.

A simple example of how this works is shown in Figure 3.1. On the first image is a visualization of the scene: two probes ( $p_0$  and  $p_1$ ) in space visualized as red dots, many receivers on scene surface as black dots and a single light source visualized as a yellow rectangle. On the second image you can see that both probes measure radiance in the scene and both of them see the light source. On the third image probes contributed measured radiance in all directions (visualized as yellow arrows). And finally on the fourth image receivers fetch irradiance from probes and accumulate it, where the amount of irradiance is visualized with yellow color. Irradiance measured in receivers is then used to illuminate scene meshes.



**Figure 3.1.** Simple visualization of rendering cycle.

Figure 3.2 shows the second iteration through the cycle. During the second rendering cycle, probes do not only see irradiance coming from the light source but also contributed irradiance in receivers. This feature is used by the system to simulate infinite light bounces.



**Figure 3.2.** Simple visualization of rendering cycle, the second iteration.

### 3.1.2 Detailed introduction

The proposed method [4] uses a variety of mathematical instruments to store and reconstruct the irradiance signal. Because they are a crucial part of the algorithm, I decided to describe them in advance.

#### 3.1.2.1 Spherical Harmonics

The proposed method [4] uses Spherical Harmonics to transport irradiance. Spherical Harmonics (SH) are functions which allow us to approximate any k-dimensional function which is defined on a surface of a sphere. The stored signal can be then reconstructed in any direction. In many ways, Spherical Harmonics are similar to Fourier transforms, as they also break the measured signal into components of some simple basis functions. In the case of Fourier transforms, those simple basis functions are sine functions. In the case of Spherical Harmonics, basis functions are noted as  $Y_l^m(\theta, \varphi)$ .

As it was noted in [16], basis functions of Spherical Harmonics are orthonormal functions over the sphere. To define a position on sphere surface, Spherical Harmonics use the following parametrization:

$$s = (x, y, z) = (\sin(\theta)\cos(\varphi), \sin(\theta)\sin(\varphi), \cos(\theta))$$

From this parametrization, basis functions are defined as:

$$Y_l^m(\theta, \varphi) = K_l^m e^{im\varphi} P_l^{|m|}(\cos\theta), l \in N, -l \leq m \leq l$$

Functions  $K_l^m$  are defined the following way:

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}$$

and  $P_l^{|m|}$  are associated Legendre polynomials.

Note, that we are going to use only real Spherical Harmonics, which is a special version of Spherical Harmonics solely defined in space of real numbers.

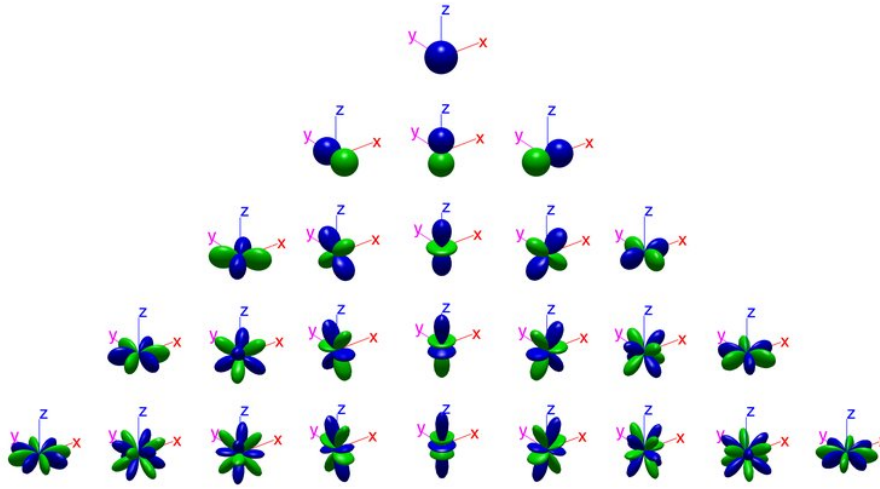
I will now show the basic usage of Spherical Harmonics. Let's say we have some measured data from some point in space, where a single data value consists of  $[\vec{v}, \vec{\omega}]$ .  $\vec{v}$  is a measured value and  $\vec{\omega}$  is a direction in which the values were measured from our point. From this set of data, we want to create a general function which would give us  $\vec{v}$  for any input  $\vec{\omega}$ . In other words, we want a function in shape  $f(\vec{\omega}) = \vec{v}$  for any  $\vec{\omega}$ . As it was mentioned in [18], to approximate such function we can use SH, with SH function  $f(\vec{\omega})$  would look something like this:

$$f(\vec{\omega}) = \sum_{l=0}^{n-1} \sum_{m=-l}^l \lambda_l^m Y_l^m(\vec{\omega}) \quad (1)$$

where  $n$  is the maximum degree of SH (sometimes called a band of SH),  $l$  goes through all degrees up to  $n$  and  $m$  through certain degree basis functions. The degree of SH crucially affects the accuracy of the reconstructed signal, the higher the degree is the better approximation is provided.  $Y_l^m(\vec{\omega})$  is a value of  $l$  degree  $m^{th}$  basis function in a direction  $\vec{\omega}$ . Basis functions of a certain degree are constant functions and their value can be computed with, for example, generator by Sloan [27], which I used in my work as well.

Here you can see, that every degree of SH has its own set of basis functions, which allows better and better approximation of the signal. An overview of SH degrees and their basis functions is in Figure 3.3. You can see that, for example, degree 0 is only able to reconstruct a single radius of values because it has only a single basis function which is equal in all directions. But degree 2 has already 9 different basis functions (degree 2 means degrees 0, 1 and 2). To obtain a number of basis functions for any degree, there is a simple equation. Let's say we have SH of degree  $N$  than this degree has  $(N + 1)^2$  basis functions.

The last coefficient in Equation (1) is  $\lambda_l^m$ , which is a unique number for this function. This number defines multiplication of basis function  $Y_l^m(\vec{\omega})$  for any  $\vec{\omega}$ . These numbers  $\lambda_l^m$  are called SH coefficients. They encode the function itself and are measured from the samples to obtain  $f(\vec{\omega})$ . Of course, the number of SH coefficients is equal to the number of basis functions.



**Figure 3.3.** Visualization of SH degrees<sup>1</sup>

An example of equations for real Spherical Harmonics functions from degree 0 to degree 2 is shown in Figure 3.4.

$Y_l^m(x, y, z)$	$m = -2$	$m = -1$	$m = 0$	$m = 1$	$m = 2$
$l = 0$			$\frac{1}{2} \sqrt{\frac{1}{\pi}}$		
$l = 1$		$\sqrt{\frac{3}{4\pi}} \cdot \frac{y}{r}$	$\sqrt{\frac{3}{4\pi}} \cdot \frac{z}{r}$	$\sqrt{\frac{3}{4\pi}} \cdot \frac{x}{r}$	
$l = 2$	$\frac{1}{2} \sqrt{\frac{15}{\pi}} \cdot \frac{xy}{r^2}$	$\frac{1}{2} \sqrt{\frac{15}{\pi}} \cdot \frac{yz}{r^2}$	$\frac{1}{4} \sqrt{\frac{5}{\pi}} \cdot \frac{-x^2 - y^2 + 2z^2}{r^2}$	$\frac{1}{2} \sqrt{\frac{15}{\pi}} \cdot \frac{zx}{r^2}$	$\frac{1}{4} \sqrt{\frac{15}{\pi}} \cdot \frac{x^2 - y^2}{r^2}$

**Figure 3.4.** Example of SH basis functions  $Y_l^m(x, y, z)$  from degree 0 to degree 2.

<sup>1</sup> <https://www.mathworks.com/matlabcentral/fileexchange/43856-real-complex-spherical-harmonic-transform-gaunt-coefficients-and-rotations> .

To simplify the equation for  $f(\vec{\omega})$ , I will use following notation (as it was used in the work [4]):

$$f_i(\vec{\omega}) = \sum_j^N \lambda_{ij} Y_j(\vec{\omega}) \quad (2)$$

where  $i$  is an index of this particular measured function and  $j$  is an index of a basis function obtained as  $j = l^2 + l + m$ . So how to compute those SH coefficients  $\lambda_{ij}$ ? A equation is as simple as final signal reconstruction, the process is just reversed:

$$\lambda_{ij} = \int_{\vec{\omega} \in \Omega} f(\vec{\omega}) Y_j(\vec{\omega}) d\vec{\omega} \quad (3)$$

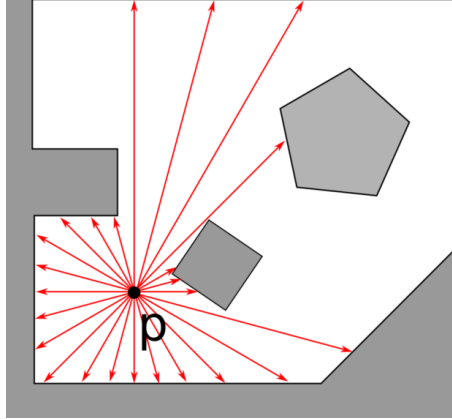
where  $\Omega$  is a set of sampled radiance.

As the set of samples  $\Omega$  is finite, computation of SH coefficients is done by Monte-Carlo integration method [28]. For more about the usage of this method in my implementation, see 3.2.3.

Once SH coefficients are known, reconstruction of the signal can be computed by Equation (2).

### 3.1.2.2 Probe irradiance measurement

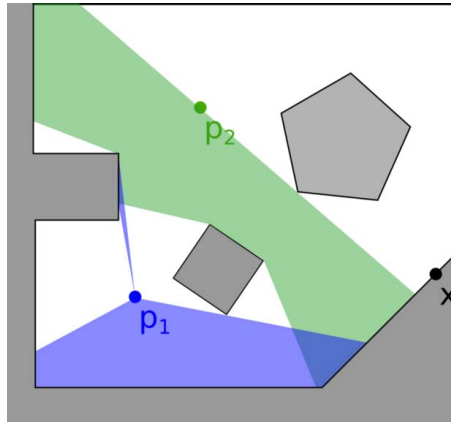
To store information about surrounding irradiance, probes use Spherical Harmonics. Firstly a probe must gather radiance samples from set  $\Omega$  to compute irradiance. This set is a precomputed set of rays (here called relight rays) and information about their hitpoints. Every probe during **the precomputation part** shoots rays uniformly distributed across the sphere (as it is shown in Figure 3.5) and stores resulting hitpoints into a data texture. During real-time rendering, the probe just measures radiance at the hitpoint and combines it with a value of appropriate basis function as it is shown in Equation (3). From these values, every probe computes its SH coefficients in real-time.



**Figure 3.5.** Visualization of probes relight rays. Casting rays (red arrows) from probe  $p$  (black dot) uniformly distributed on a sphere.

### 3.1.2.3 Receiver irradiance contribution

A contribution of irradiance in a receiver is done by gathering radiance from probes SH functions. This gathering must be done in such a way, that only radiance which is able to reach our receiver is gathered. Visualization of this selection is shown in Figure 3.6. Here we have two probes  $p_1$ ,  $p_2$  and receiver  $x$ . The receiver must gather radiance from surfaces that are visible to it. For example, if the receiver does not see some light source, it cannot gather radiance from that light source. As radiance has been measured only in probes, probes are our only source of irradiance gathering. Thus we must combine these visibility rules to one: Receiver  $x$  gather irradiance from probe  $p_i$  in direction  $\vec{\omega}$ , if both the probe and the receiver see in that direction exactly the same surface. You can see a visualization of this visibility match between probes  $p_1, p_2$  and receiver  $x$  in Figure 3.6.



**Figure 3.6.** Visualization of visibility match between probes (blue  $p_1$  and green  $p_2$ ) and a receiver  $x$ .

To store information about this visibility match for each receiver in the work [4] they introduced the following visibility function:

$$K_{ij}(x, \vec{\omega}) = \frac{w_i(x) V_i(\vec{\omega}) Y_j(\Psi(\vec{\omega}))}{\sum_k w_k(x) V_k(\vec{\omega})}$$

where  $w_i(x)$  is a weight function between probe  $i$  and receiver  $x$  defined as  $w_i(x) = w(\frac{\|x - p_i\|_2}{r})$ , where  $p_i$  is a position of the probe,  $r$  is a constant radius of probes (single number which will be computed at the beginning of the precomputation part) and  $w(t) = 2t^3 - 3t^2 + 1$  for  $t \in [0, 1]$ .  $\vec{\omega}$  is a direction from the receiver to a surface visible to the receiver,  $\Psi(\vec{\omega})$  is direction from the probe to the same surface.  $V_i(\vec{\omega})$  is a binary function,  $V_i(\vec{\omega}) = 1$  if the surface visible to receiver  $x$  from the direction  $\vec{\omega}$  is also visible to probe  $i$ .

Function  $K_{ij}(x, \vec{\omega})$  encodes not only how much from probe  $i$  is visible to receiver  $x$ , but this visibility is also weighted by their distance. This function can be then used to compute a visibility match from any direction  $\vec{\omega}$ . The visibility match is in the work [4] noted as  $\alpha_{ij}$  and can be computed as an integral from  $K_{ij}(x, \vec{\omega})$  for all  $\vec{\omega}$ , as it is shown in the following equation:

$$\alpha_{ij} = \int_{\vec{\omega} \in \Omega} K_{ij}(\vec{\omega}) d\vec{\omega} \quad (4)$$

Gathered irradiance in some receiver  $x$  is computed in the real-time using the following equation:

$$I(x) \approx \int P_x\{\lambda\}(w) \cos\theta d\vec{\omega} = \sum_{ij} \lambda_{ij} \alpha_{ij} \quad (5)$$

where  $i$  is an index of a probe. Notice similarities with Equation (2), where  $\alpha_{ij}$  can be seen as  $\forall \vec{\omega} : Y_j(\vec{\omega})$  enriched via weighted visibility.

The reasoning of integration in Equation (4) is the following: We know that gathered irradiance in receiver  $x$  can be computed by Equation (5). Now if I rewrite exactly the same equation just using  $K_{ij}(\vec{\omega})$  it will look like this:

$$I(x) \approx \int_{\vec{\omega} \in \Omega} \sum_{ij} \lambda_{ij} K_{ij}(\vec{\omega}) d\vec{\omega} \quad (6)$$

This equation essentially just choose a direction  $\vec{\omega}$ , calculate an amount of incoming irradiance from that direction and aggregate it to  $I(x)$ , then choose another direction  $\vec{\omega}$  and so on. But coefficients  $\lambda_{ij}$  are not dependant on the integral over all directions  $\Omega$ . So we can rearrange the equation to this state:

$$I(x) \approx \sum_{ij} \lambda_{ij} \int_{\vec{\omega} \in \Omega} K_{ij}(\vec{\omega}) d\vec{\omega} \quad (7)$$

From which is clearly visible Equation (4). Values  $\alpha_{ij}$  are precomputed for all receivers and stored to be used in real-time rendering. Computation of  $\alpha_{ij}$  in the algorithm can be found in section 3.2.3.

## 3.2 Precomputation Phase

During the precomputation part, the algorithm precomputes all data required for real-time rendering. The precomputed data are then stored in data textures. The implementation details of the precomputation application are described in this chapter.

### 3.2.1 Irradiance Receivers Placement

Irradiance receivers should cover the whole scene in such a way that every object (mesh) of the scene has an equal density of receivers. It is also desirable to avoid placement of receivers inside of the geometry or to areas, where should be always dark (contact points of meshes). In the following chapters I will describe how I implemented this process to be fully automatic.

#### 3.2.1.1 Lightmap UV Coordinates for Meshes

In order to calculate irradiance of all meshes we have to place them on the lightmap texture, so every mesh will have its own UV coordinates for user-defined textures and UV coordinates for the lightmap texture. My goal was to place all meshes UV coordinates on a single texture, so that the texture is square, no mesh UV coordinates collide with another mesh UV coordinates and the texture should have minimum empty space. The minimum empty space constraint yields 2D Knapsack problem which is NP hard problem. To solve this problem in a polynomial time I designed an approximation algorithm, which I will now describe. All UV coordinates are normalized, but not likely all meshes should cover the same area in the lightmap texture. To determine how much area each mesh should cover, I calculated a mesh scaling factor. I define the mesh scaling factor in the following way:

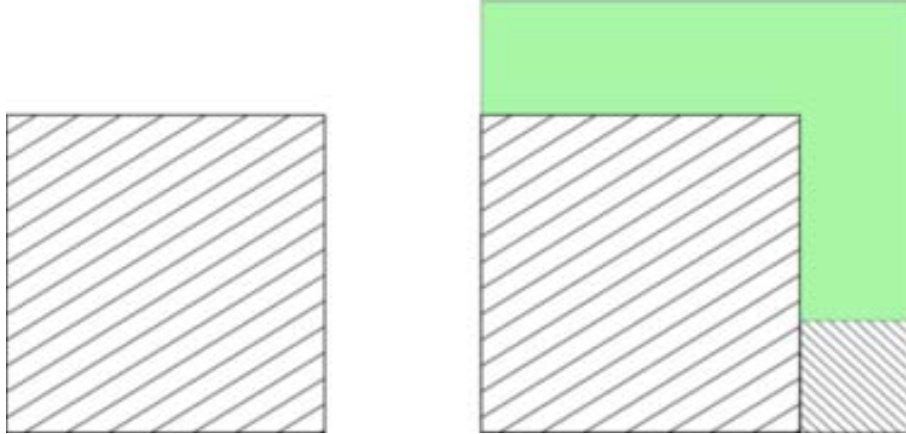
Let's take an edge  $e$  from the mesh in 3D and it's UV coordinates correspondent  $j$ . Edge  $e$  has two 3D points  $\vec{x}$  and  $\vec{y}$ , edge  $j$  has two 2D points  $\vec{x}_{uv}$  and  $\vec{y}_{uv}$ , where  $\vec{x}_{uv}$  are the UV coordinates of point  $\vec{x}$  and  $\vec{y}_{uv}$  are the UV coordinates of point  $\vec{y}$ . The scaling factor is then defined as  $\frac{\sum_{e \in E} \frac{\|\vec{e}\|}{\|\vec{j}\|}}{\|E\|}$ , where  $E$  is a set of all edges from the current mesh. Note that averaging over all edges is not necessary if the unwrapped model is not deformed. The scaling factor will be relative to the area this mesh should cover in the lightmap. To ease the problem, firstly I approximate the mesh in a texture space by a quad of his texture space. So it is a quad with size  $[0, 1]^2$ . I sort all meshes by their scaling factor in decreasing order and place a mesh with the biggest factor in a canvas (which will be our lightmap texture after normalization of UV coordinates).



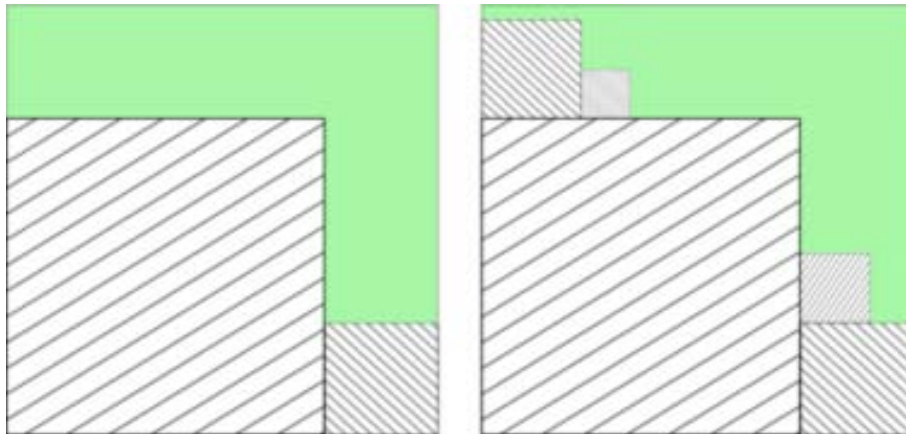
Then I place the rest of the meshes using the following rules:

- 1) If there is no free space, place the biggest (not yet placed) mesh into the canvas to the bottom right corner, upscale size of the canvas and mark free space.
- 2) If there is free space available, place the biggest mesh which fits into this space and marks free space.

Rules are visualized in figures 3.7 and 3.8.

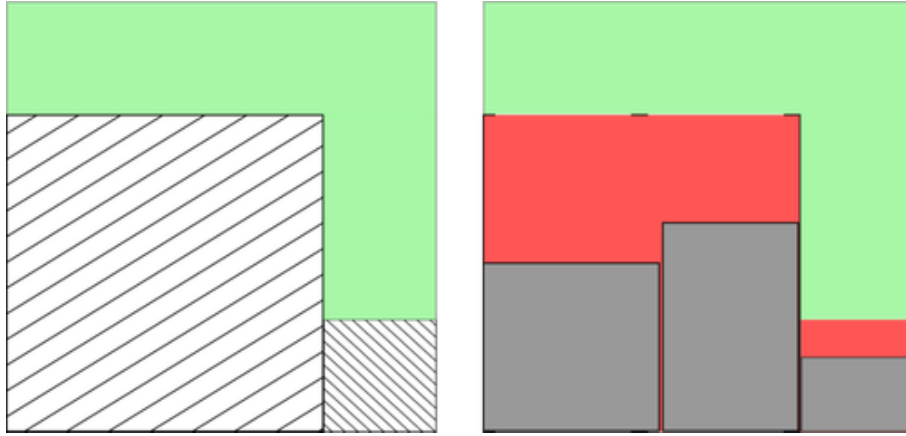


**Figure 3.7.** Lightmap UV coordinates calculation. The first rule: on the left is canvas before, on the right is canvas after placement, the green area is free space available.



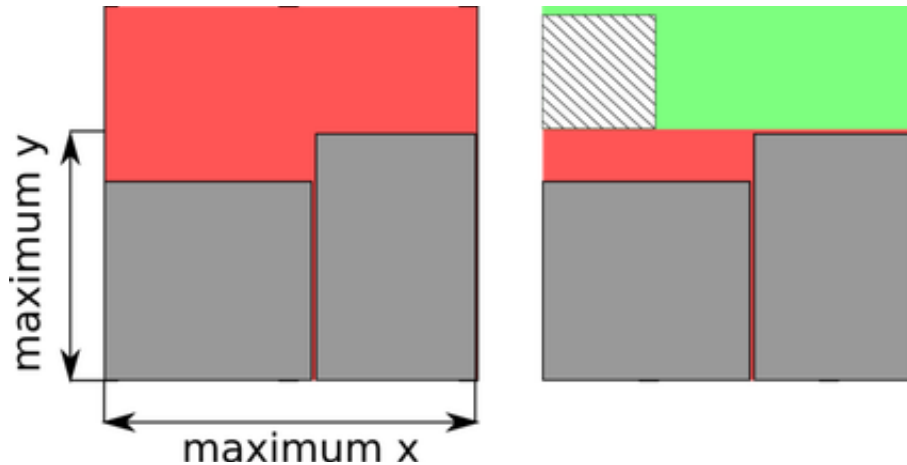
**Figure 3.8.** Lightmap UV coordinates calculation. The second rule: on the left is canvas before, on the right is canvas after placements, free space is always filled from the bottom left corner.

There can still be a lot of unused space if mesh itself does not cover its UV coordinate system regularly, as it is visible in Figure 3.9.



**Figure 3.9.** Lightmap UV coordinates calculation, unused space visualization. On the right picture is visualized unused free space with red color.

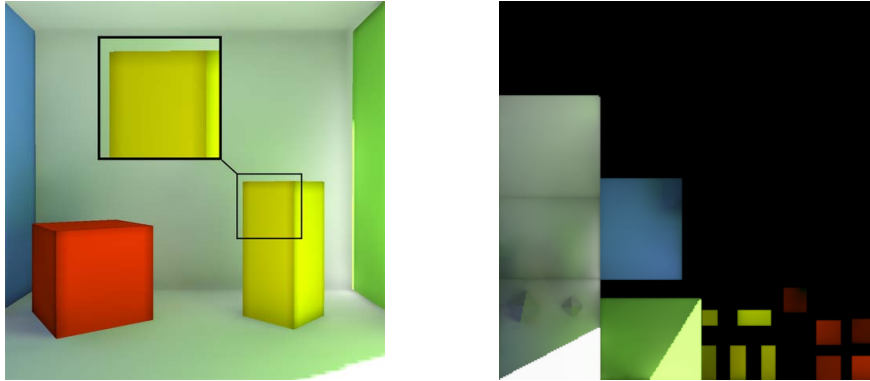
To use this free space, I measure the maximum  $x$  coordinate and the maximum  $y$  coordinate of this particular UV grid. A rectangle of free space is then a gap between those maximal coordinates and the maximum of maximal coordinates as it is shown in Figure 3.10.



**Figure 3.10.** Lightmap UV coordinates calculation, filling of unused space. Free space is colored with green color, another mesh UV grid is placed in new free space.

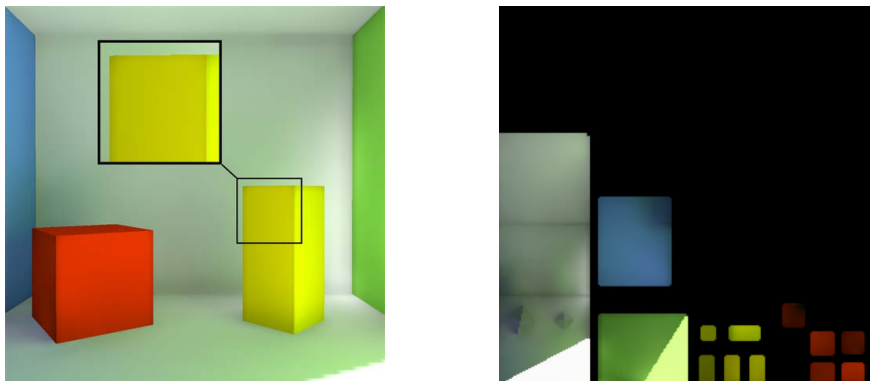
After placing all meshes into the canvas I normalized their new UV coordinates with a final size of the canvas. This solution has one issue. Using this method, neighboring meshes have no space between them. This is a problem because I am using linear texture filtering for smooth irradiance interpolation during real-time rendering.

Two neighboring meshes may then affect each other, as it is shown in Figure 3.11. Yet another problem is visible from this figure as well. Due to the linear texture filtering, there is blending between an initial black background color of the lightmap texture and the actual irradiance on the mesh. This causes edges of meshes to be dark.



**Figure 3.11.** Incorrect UV coordinates. On the left: rendered scene, notice dark edges of meshes. On the right: the lightmap texture, no borders between objects.

To solve both of these problems, I set a constant gap between all meshes in the lightmap texture to three pixels. Every receiver which is on the edge of a mesh in the lightmap texture will be extending its calculated irradiance one pixel behind the mesh border. This will cause correct linear filtering of the irradiance texture as it is shown in Figure 3.12.



**Figure 3.12.** Corrected UV coordinates. On the left: rendered scene. On the right: the lightmap texture, gaps between objects and single pixel borders.

### 3.2.1.2 Receiver Placement

It only makes sense if such a receiver is on a surface of some object because receivers of irradiance are texels of the lightmap. For every receiver, I need its position in 3D, a normal vector, coordinates of the texel and a material of a mesh where the receiver is. I used OpenGL to gather this information. Using UV coordinates of the lightmap texture as a position coordinates I rendered 3 different textures: one texture with interpolated 3D positions, one with interpolated normal vectors and one with an ID of a material which certain mesh uses. Receivers were then gathered from these three textures. Texels of these textures were used to create a receiver if a normal vector at texel coordinates was not zero.

### ■ 3.2.2 Probe Placement & Radius Calculation

Probes have similar requirements for their placement as receivers, they also should cover the scene with constant density. Probes, however, are placed in the space, not on the surface of the scene. It is also crucial to not place probes inside of the geometry as this would possibly lead to faulty results. Implementation details about the placement and probe radius calculation are provided in this chapter.

#### 3.2.2.1 Probe Placement

Firstly I created a voxel grid over the whole scene. A size of one voxel in every axis is dependent on constant  $\rho_p$  (as it was named in the work [4]). After the voxel grid is created and all objects are assigned to appropriate cells, I marked all voxels where probes could be. In the work [4], they used a flood-fill algorithm to obtain this list of voxels. I used a different method. There are basic rules which I used to determine the correct voxels:

- 1) The probe should not be in the same voxel with another object.
- 2) The probe should be in the voxel near a surface of some object.
- 3) The probe should not be inside an object.

I used a simple algorithm to determine correct voxels with these rules in mind. I firstly calculated a normal vector of every voxel, which was not empty. The normal vector was determined by two rules described in Figure 3.13. The green point indicates a voxel where the probe could be placed. If a voxel contains more than a single object, I determine (via dot product) if normals of those objects point in roughly the same direction. If they do, the voxel still has the normal vector. If not, the voxel loses the normal vector and voxels near him cannot contain probes. The last rule can be changed so that a voxel, which does not have a normal vector, can have probes on both sides. This would produce more probes, but would solve problematic (for example really thin) meshes.



**Figure 3.13.** Voxel normal consistency check. On the left: the red arrow is a normal vector of the object, the blue arrow is a normal vector of the voxel. On the right: no normal vector of the voxel.

As a next step I determined the desired number of probes. This number can be calculated as a number of cells in a grid with spacing set to a value of constant  $\rho_p$  which covers the whole scene. Let us suppose a scene with axis-aligned bounding box (AABB) as its bounding volume (BV), let  $x_{size}$ ,  $y_{size}$ ,  $z_{size}$  be a size of this AABB in x-axis, y-axis and z-axis. If we want to calculate a number of cells in a grid with a cell size set to  $\rho_p$ , we compute the number of cells in every direction and multiply them together:

$$Desired\ probe\ count = \frac{x_{size} \times y_{size} \times z_{size}}{\rho_p}$$

The final step was lowering a number of probes to the desired count. We want to have probes equally scattered in the scene. Thus, we need to have a constant density of probes in the scene.

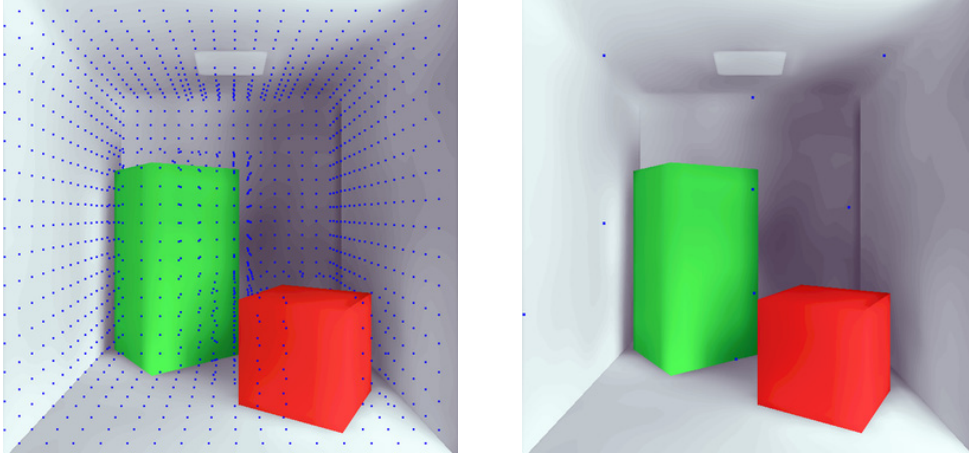
Firstly I introduce a density calculation as it was written in the work [4]. They use a spacial weight kernel  $w_i(x, r) = w(\|x - p_i\|_2/r)$  in this form:

$$w(t) = 2t^3 - 3t^2 + 1, \text{ if } 0 \leq t \leq 1; 0 \text{ otherwise}$$

where  $r$  is some particular radius. In our case, we will calculate the density of a certain probe  $p_i$  as the following:

$$K_i = \sum_j w_i(p_j, \rho_p)$$

where probes  $p_j$  are all probes except probe  $p_i$ . I calculate this kernel density for all probes, then I remove a probe with the highest density and iteratively continue until I have exactly as many probes as it was desired. In Figure 3.14 you can see the initial set of probes created directly from the voxel grid and on the right-hand side is the final probe set.



**Figure 3.14.** Probe positioning (probes visualized as blue dots). On the left: an initial dense set of probes from a voxel grid. On the right: final probe set with similar kernel density.

### 3.2.2.2 Probe Radius

The last step is radius calculation. Here comes in place a user-defined constant  $N\_OVERLAPS$ , which defines how many probes should support each receiver (for all examples I used  $N\_OVERLAPS = 10$ ). Probe supports only those receivers who are within a radius of that probe. The radius is unknown but from  $N\_OVERLAPS$  constraint we can figure this number out. To avoid computation of radius using all receivers, I chose smaller set of sample receivers. For each of these sample receivers I calculate a radius  $r_i$ . Radius  $r_i$  is such a number for receiver  $i$ , that  $N\_OVERLAPS$  closest probes have this receiver under their support. The probe radius is then the average of these radii. Note that we have to calculate only a single radius for all probes due to a similar density of probes in the scene.

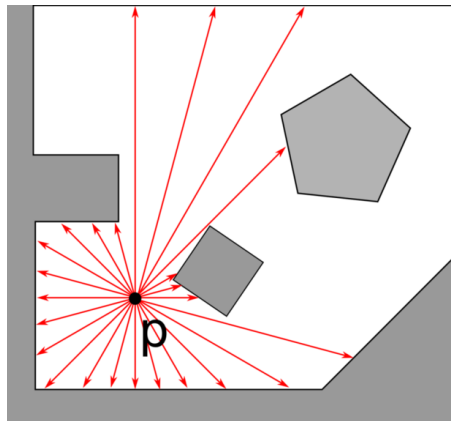
### ■ 3.2.3 Probe Ray-Casting & Receiver Coefficients

Probes need to know where to look to the lightmap for irradiance. For this purpose in the proposed method [4] they used precomputed hitpoints from ray-casting. Receivers, however, does not have to look into lightmap at all as they are gathering irradiance from probes. Thus, we can simply precompute all visibility and weights for receivers in advance. Details about these topics are covered in the following sections.

#### 3.2.3.1 Probe Ray-Casting

For a contribution of radiance in probes in the work [4] they use precomputed ray-casting. I implemented the same method. The same way as it was mentioned in the method overview, I cast from every probe a set of rays which are uniformly distributed across a sphere (see Figure 3.15). For each ray, I mark information about a material index of the mesh the ray hit, a UV lightmap coordinates of the hitpoint and a 3D coordinates of the hitpoint. If the ray hits nothing I set its material to some predefined error value. This is the part where environment mapping can come in place. If a user wants to use an environment map to illuminate meshes, he / she needs only to set correct UV coordinates to rays which hit nothing during casting.

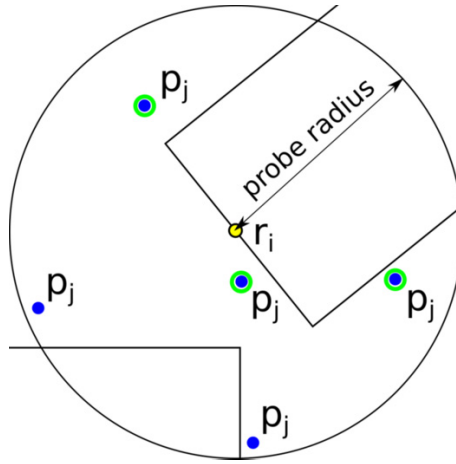
Information collected during ray-casting will be later stored in data texture called **Probe Ray Intersection texture**. See section 3.2.5 for details about this data texture.



**Figure 3.15.** Probe Ray-casting, casting rays (red arrows) from probe  $p$  (black dot) uniformly distributed on a sphere.

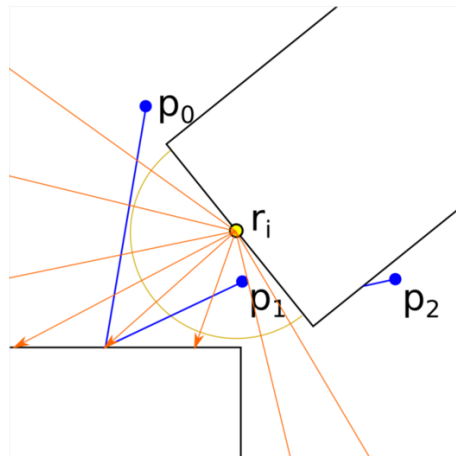
### 3.2.3.2 Receiver Coefficients

Suppose we are going to calculate coefficients for receiver  $r_i$ . We need to determine which probes will contribute their radiance to our receiver  $r_i$ . To do so, I measured a distance from  $r_i$  to every probe  $p_j$ . If the distance is smaller than our measured probe radius, this probe  $p_j$  will contribute its radiance to the receiver. This way I determine  $N\_OVERLAPS$  probes which distances were closest to receiver  $r_i$  (note that  $N\_OVERLAPS$  is the maximal number of supporting probes). See visualization in Figure 3.16. Notice that receiver  $r_i$  does not have to directly see probe  $p_j$  to be under its support. If no probe supports this receiver within the probe radius, we can discard this receiver completely because no radiance will ever reach it (as probes are the only source of irradiance gathering).



**Figure 3.16.** Probes within radius support receiver  $r_i$ ,  $N\_OVERLAPS = 3$

With the list of supporting probes, I start casting rays distributed over a hemisphere (which is rotated in the direction of the receiver normal vector). For each ray I determine if any probe sees a hitpoint of that ray as it is visualized in Figure 3.17 (only one hitpoint is traced for the visualization).



**Figure 3.17.** Receiver, casting rays (orange color), tracing hitpoint with rays from probes (blue color).

If probe  $p_j$  sees hitpoint, we will mark its radiance contribution from this direction. When we need to determine irradiance at  $r_i$  we will take into account radiance at  $p_j$  in the direction from probe  $p_j$  to this hitpoint.

As it was covered in section 3.1.2, in the work [4] they introduce this equation to compute a receiver irradiance:

$$I(x) \approx \int P_x\{\lambda\}(w) \cos\theta d\omega = \sum_{ij} \lambda_{ij} \alpha_{ij} \quad (8)$$

where we are now computing  $\alpha_{ij}$ . We can express  $\alpha_{ij}$  like  $\langle K_{ij}(x, \cdot), \Phi \rangle$ , which is a weighted visibility function over the whole sphere (every direction  $\Phi$ ), defined as:

$$K_{ij}(x, \omega) = \frac{w_i(x) V_i(\omega) Y_j(\Psi(\omega))}{\sum_k w_k(x) V_k(\omega)}$$

$\alpha_{ij}$  is  $K_{ij}(x, \omega)$  for all possible  $\omega$ , to obtain such a number I used the following integral:

$$\alpha_{ij} = \int_{\omega \in X} K_{ij}(\omega) dX \quad (9)$$

The reasoning for this integration has been covered in section 3.1.2.

To approximate  $\alpha_{ij}$  from samples of  $K_{ij}$  gathered from ray-casting, I used Monte-Carlo integration:

$$\alpha_{ij} \approx \frac{1}{N} \sum_{\omega} K_{ij}(\omega) w(\omega)$$

where  $w(\omega)$  is the weight of the sample. The weight of the sample is inverse of a probability of the sample. In this case, every sample of integration has the same weight equal to  $2\pi$  ( $2\pi$  because we use only half of sphere). So the final integration equation is:

$$\alpha_{ij} \approx \frac{2\pi}{N} \sum_{p \in P} K_{ij}(ray_p)$$

where  $P$  is a set of rays. Note that  $K_{ij}$  contains information about Spherical Harmonics in a direction of  $ray_p$ . This means, that in the direction from probe  $p_j$  to a hitpoint we evaluate Spherical Harmonics and add this weighted evaluation to our receiver coefficients.

Question is, what to do with rays from receiver  $r_i$  which hit nothing during ray-casting. This topic has been covered in chapter 5.1.3.



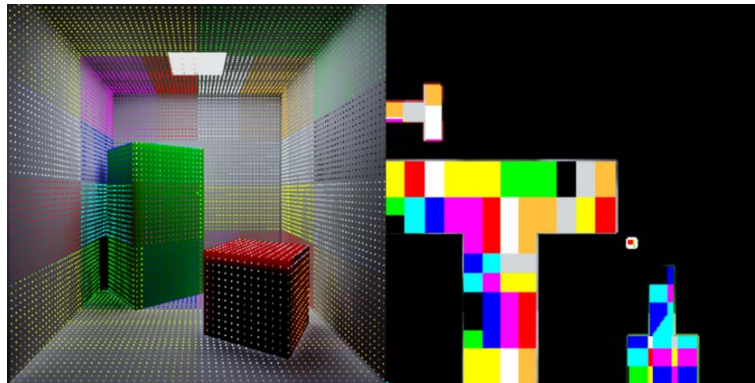
### 3.2.4 Receiver Clustering

A number of irradiance transport coefficients  $\alpha_{ij}$  for each receiver in a scene is huge. It is rather inefficient to store all these coefficients separately. To solve this problem, in the work [4] they used PCA to reduce the number of coefficients needed to transport irradiance. Furthermore, they used Clustered PCA (CPCA) because the calculation of SVD for large matrices is not effective as well.

Now I will describe the solution they proposed and I implemented. I divided the scene into smaller clusters. This division has been made by splitting a scene's AABB into half (always splitting the largest dimension). If a certain leaf of this AABB tree has less than 1024 receivers, I calculated PCA matrices for this cluster (I will describe a particular shape of these matrices later). If the PCA projection was sufficient (an error of projection was less than 0.005), I accepted this composition and stopped division in this leaf. Now I will describe it in more details.

#### 3.2.4.1 Scene AABB Tree Division

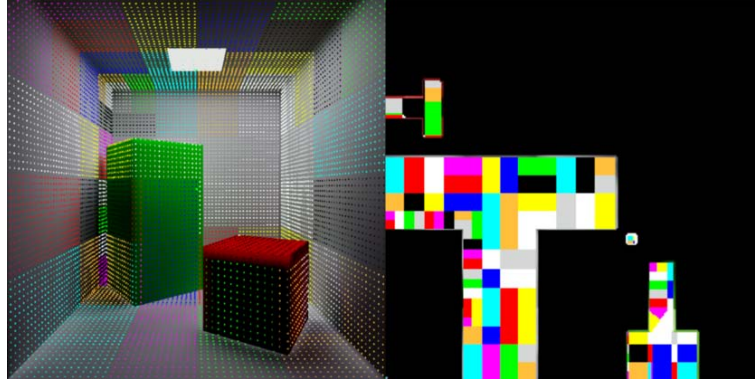
I created an AABB over the whole scene. This was my starting point of the division. In every next step I chose the widest axis of this AABB and split it in half along this axis. If the resulting AABB has less than 1024 receivers, I created PCA matrices. To visualize, how this threshold-based refinement affects the final image and a scene clustering, I computed a single scene with two different setups. The first scene was computed without any threshold-based refinement, the second scene with threshold-based refinement and the threshold was set to 0.0001. The threshold was set this low (in the work [4] and in rest of the renders I use threshold 0.005) to emphasize splitting during threshold-based clustering. The resolution of the lightmap texture was set only to  $256 \times 256$  to emphasize splitting and support incoherence of the scene (receivers are sparsely covering the scene, so they have less similar irradiance transport coefficients). Visualization of clustered receivers is visible in Figure 3.18. Here is the AABB split only based on a number of receivers. On the left you can see receivers visualized as colored dots in 3D render, on the right they are placed in the lightmap texture.



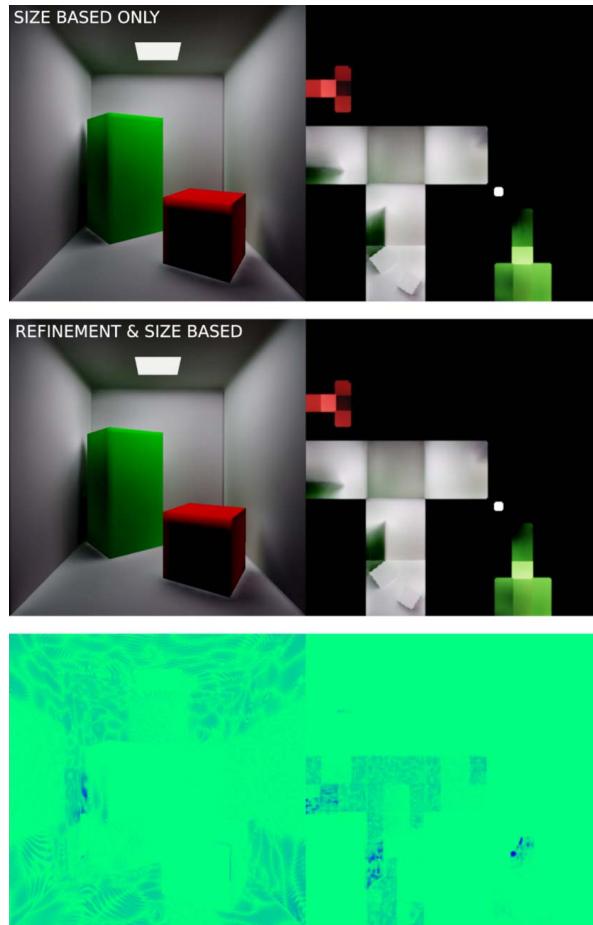
**Figure 3.18.** Clustering visualization without threshold-based splitting, on the left: 3D render with receivers visualized as colored dots, on the right: receivers visualized on lightmap texture. 24 clusters in total.

This method does not provide good results as the accuracy of the projection via SVD is different in every cluster in the scene. To solve this problem, in the paper [4] they measure the error of projection.

Based on this error, they decided if the splitting of the AABB should continue or not. After applying this rule to my implementation the number of clusters increased but it provided much better results. Visualization of the clustering is in Figure 3.19, a comparison of renders rendered without and with the threshold-based splitting is in Figure 3.20. The comparison was calculated using Structural Similarity index (SSIM).



**Figure 3.19.** Clustering visualization with threshold-based splitting, on the left: 3D render with receivers visualized as colored dots, on the right: receivers visualized on lightmap texture. 43 clusters in total.



**Figure 3.20.** Clustering comparison. On the left are 3D renders, on the right are lightmap textures. The first row is rendered with size-based refinement only, middle row is with threshold-based and size-based refinement and the last row is an SSIM comparison of both renders.

### 3.2.4.2 Cluster PCA Matrices & Error Computation

Now into more details about the PCA matrices used in the implementation. Irradiance in receiver  $\mathbf{r}$  can be computed with the following equation:

$$I(r) \approx \int P_x\{\lambda\}(w) \cos\theta d\omega = \sum_{ij} \lambda_{ij} \alpha_{ij}$$

As it was mentioned before, the number of  $\alpha_{ij}$  coefficients is huge. To solve memory and computation time problems, PCA was used. During the creation of a certain cluster, a list of receivers pertaining to this cluster is received. Let us call this list  $\mathbf{R}$ . Every receiver has list of probes which support him. From these lists I create single list of unique probes for this cluster, which consists of all probes supporting receivers in this cluster (this list can be in any order, but the order matters). Every probe from this list has its SH coefficients  $\lambda_{ij}$ . We just need to match them with appropriate  $\alpha_{ij}$  of a certain receiver. As an example I provide visualization in Figure 3.21. Here our transport matrix is called  $\mathbf{A}$ . In this particular case, we used 4 coefficients (SH degree 1) to compute irradiance. There are 3 unique probes supporting this cluster in order: (2, 4, 7). You can notice, that receiver  $x_1$  is supported by all probes, so its line in matrix  $\mathbf{A}$  (the first line) is full of non-zero coefficients  $\alpha_{ij}$ . But receiver  $x_4$  is supported only by probe 7. Thus, the fourth line in  $\mathbf{A}$  is mostly zero except for coefficients of probe 7.

$\mathbf{A} \quad \lambda = \mathbf{l}$

$\alpha_{02}$	$\alpha_{12}$	$\alpha_{22}$	$\alpha_{32}$	$\alpha_{04}$	$\alpha_{14}$	$\alpha_{24}$	$\alpha_{34}$	$\alpha_{07}$	$\alpha_{17}$	$\alpha_{27}$	$\alpha_{37}$
$\alpha_{02}$	$\alpha_{12}$	$\alpha_{22}$	$\alpha_{32}$	0	0	0	0	0	0	0	0
0	0	0	0	$\alpha_{04}$	$\alpha_{14}$	$\alpha_{24}$	$\alpha_{34}$	0	0	0	0
0	0	0	0	0	0	0	0	$\alpha_{07}$	$\alpha_{17}$	$\alpha_{27}$	$\alpha_{37}$

$=$ 

$\lambda_{02}$
$\lambda_{12}$
$\lambda_{22}$
$\lambda_{32}$
$\lambda_{04}$
$\lambda_{14}$
$\lambda_{24}$
$\lambda_{34}$
$\lambda_{07}$
$\lambda_{17}$
$\lambda_{27}$
$\lambda_{37}$

$l(x_1)$
$l(x_2)$
$l(x_3)$
$l(x_4)$

**Figure 3.21.** Visualization of initial transport matrix  $\mathbf{A}$ .

Now we will apply Singular Value Decomposition to obtain three matrices:

$$A = U\Sigma V^T$$

In the paper [4], they truncated this decomposition (left only 32 biggest singular values from matrix  $\Sigma$ ). This approximation saves memory but can crucially devastate irradiance transport. To prevent this from happening, we will measure an error caused by this approximation. The error of SVD projection can be calculated as:

$$1.0 - \sqrt{\frac{\sum_{i=1}^{32} s_i}{\sum_j s_j}}$$

where  $s_i$  is the  $i$ -th biggest singular value from the matrix  $\Sigma$ . As it was written in the paper [4], I accept all decompositions with the error of projection less than 0.005.

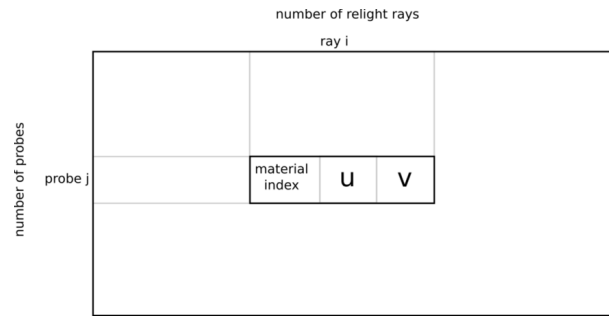
Matrix  $\Sigma V^T$  will be stored in **Cluster Matrix texture**, see section 3.2.5 for details about this texture. Matrix  $U$  will be stored in **Receiver Transport texture**, see section 3.2.5 for further details.

### 3.2.5 Computational Textures and Output Data File

Results from the precomputation application are stored in an SSF file which is a binary file containing all information about the static scene, its textures, constants etc. Essentially, the whole scene can be rendered just using data from this single SSF file. In my implementation of the method [4] I used 8 major data textures for real-time computations. I will now describe all of them.

#### 3.2.5.1 Probe Ray Intersection Texture

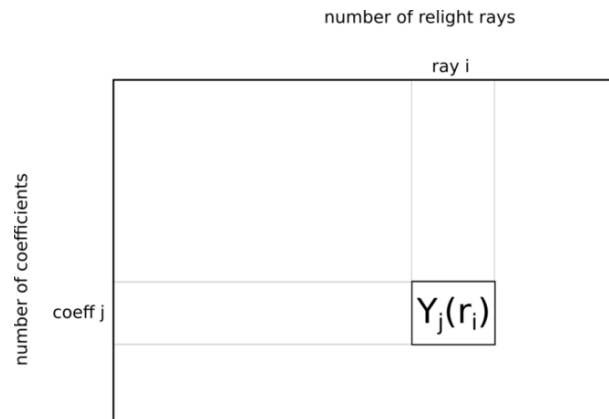
This texture is used to store information about ray hits traced during precomputation. The texture is visualized in Figure 3.22. A single item, being intersection information from ray  $i$  of probe  $j$ , of this texture, consists of three values. The first value is an index of a material, which the intersected object has. The second and the third values are UV coordinates to the lightmap texture. These coordinates correspond to the hitpoint of ray  $i$ .



**Figure 3.22.** Ray Intersection texture with single item at position  $(i,j)$ .

#### 3.2.5.2 Spherical Harmonics Texture

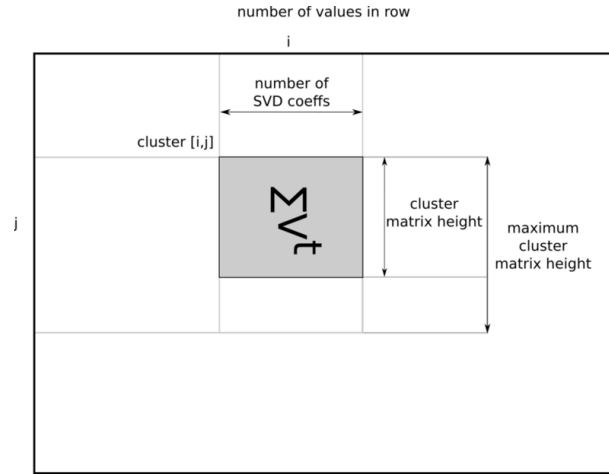
This texture stores precomputed values of Spherical Harmonics basis functions used to calculate SH coefficients in real-time rendering. Visualization of the texture is in Figure 3.23. A single item of this texture consists of one float value which represents a calculated value of a coefficient  $j$  in the direction of a ray  $i$  (noted as  $Y_j(r_i)$ ).



**Figure 3.23.** Spherical Harmonics texture with a value of a coefficient  $j$  in the direction of a ray  $i$ .

### 3.2.5.3 Cluster Matrix Texture

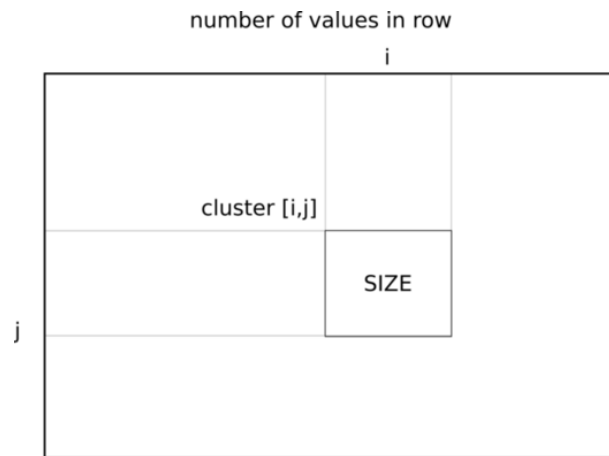
Cluster Matrix texture stores all CPCA transport matrices for receiver clusters. See Figure 3.24 which represents a visualization of this texture. Clusters are indicated with two-dimensional indexes. This decision has been made to fit big textures to GPU memory. In all my renders I used `number of values in a row` set to 128, `number of SVD coeffs` to 32 (this is equal to the number of singular values used in the truncated transport matrix). Value `maximum cluster matrix height` depends on the input scene. This texture can be a limiting factor of the algorithm due to the huge number of clusters and sizes of their transport matrices.



**Figure 3.24.** Cluster Matrix texture with an item equal to CPCA transport matrix.

### 3.2.5.4 Cluster Size Texture

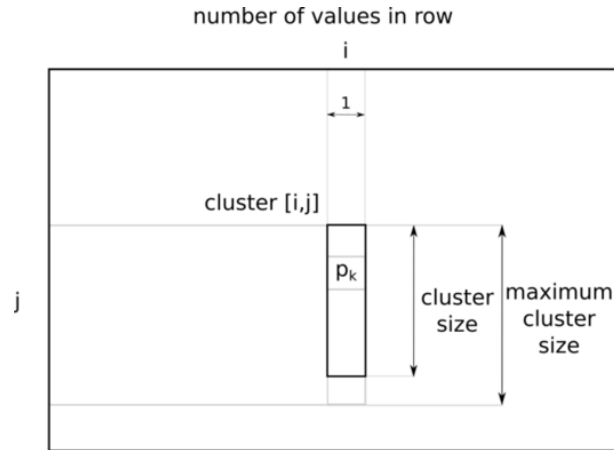
This simple texture consists of a single integer per item. The integer represents the size of a cluster. The size of the cluster is defined as a number of probes which contribute their radiance to this cluster. The value `cluster matrix height` is dependent on the size of the cluster in the following way: `cluster matrix height = cluster size × number of coefficients`, where `number of coefficients` is a number of Spherical Harmonic coefficients (I used an order 7 of SH, so that is 64 coefficients). A visualization of the texture is in Figure 3.25.



**Figure 3.25.** Cluster Size texture with an item equal to CPCA size of cluster.

### 3.2.5.5 Cluster Probe Index Texture

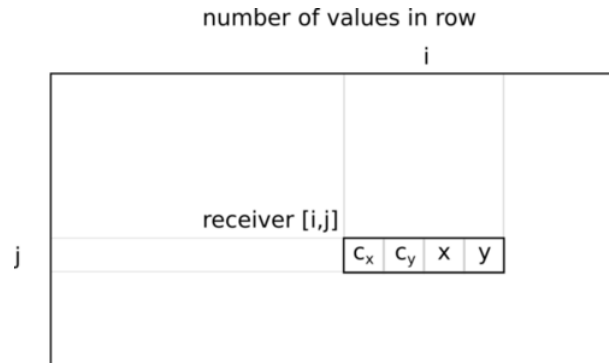
Cluster Probe Index texture is visualized in Figure 3.26. Again, clusters are indexed the same way they were in Cluster Matrix texture and Cluster Size texture. Here, each item is a vector of indexes, where each index points to a probe which supports this cluster. Probes are in the same order as the corresponding matrix in Cluster Matrix texture was created. You can see that `cluster size` from the previous texture is equal to the length of the probe-index vector.



**Figure 3.26.** Cluster Probe Index with an item equal to a vector of indexes of cluster-supporting probes.

### 3.2.5.6 Receiver Map Texture

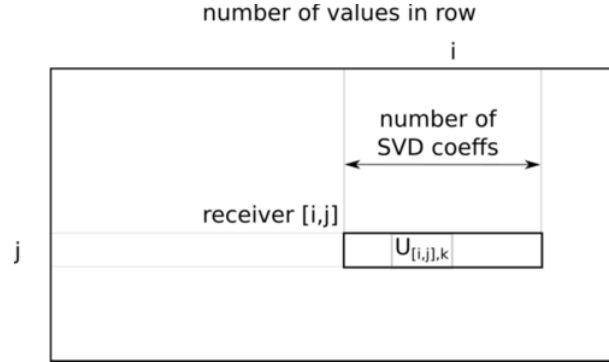
This texture is visualized in Figure 3.27. Single item consists of 4 values:  $[c_x, c_y]$  is a cluster index to which the receiver belongs and  $[x, y]$  is a position of a pixel in the lightmap texture, where this receiver will write its irradiance.



**Figure 3.27.** Receiver Map texture with an item equal to coordinates of a cluster and coordinates of an appropriate texel in the lightmap texture.

### 3.2.5.7 Receiver Transport Texture

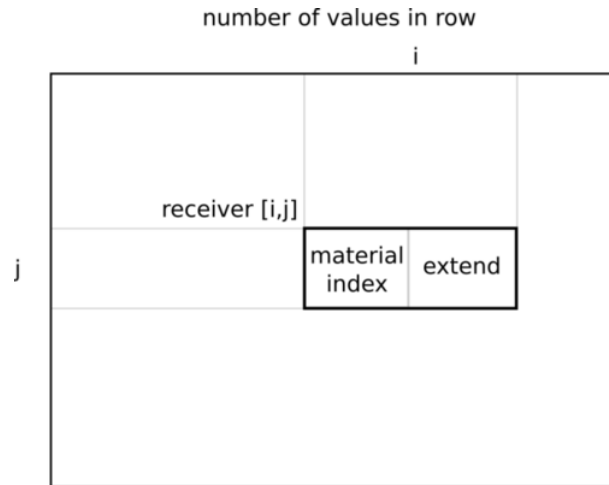
This texture is the second part of CPCA matrices. Visualization is provided in Figure 3.28. One item of this texture is equal to a row in a matrix  $U$  from SVD,  $A = U\Sigma V^t$ . This item is a transport vector which transforms an incoming cluster basis to irradiance in receiver  $[i, j]$ .



**Figure 3.28.** Receiver Transport texture with an item equal to a transport vector from SVD.

### 3.2.5.8 Receiver Material Map Texture

This texture is visualized in Figure 3.29. Single item consists of two values. The first value is an index of a material of a mesh, where the receiver lies. This information is used for illumination of emissive materials. The second value **extend** is used when the receiver is on the edge of mesh UV grid to extend irradiance by a single pixel as it was mentioned in Receiver Placement section (3.2.1). This value is bitwise read in GPU. The first four bits represent four neighboring pixels to extend. When no bit is set then no pixel is extended.



**Figure 3.29.** Receiver Material Map.



### 3.2.5.9 The Rest of Textures

For computation, few more textures are needed. However, they are not as important. Most of them are used only for my extensions, see chapter 4. I will briefly describe which textures are computed and used in my work.

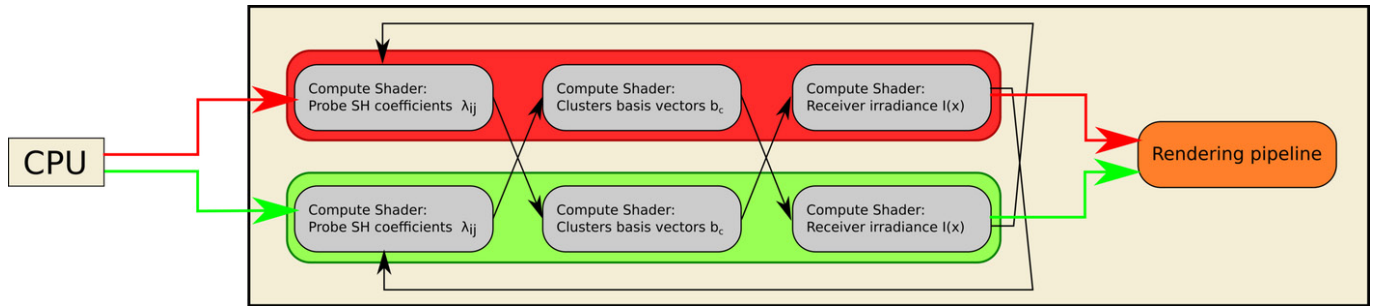
- 1) Probe position texture  
Stores position of probes in space, used for method extension (LOD).
- 2) Probe Hit Point position texture  
Stores position of hit points from probes relight rays in space, used for method extension (LOD).
- 3) Receiver position texture  
Stores position of receivers in space, used for method extension (LOD).
- 4) Cluster Bounding Volume texture  
Stores 8 points of each cluster AABB in space, used for method extension (LOD).

### 3.3 Rendering in Real-Time

For real-time rendering, I chose OpenGL API together with few extensions. At the beginning, the application loads an input SSF file and set every texture, constant, etc. as it was written in that SSF file. Now I will describe the main rendering loop.

#### 3.3.1 Main Rendering Loop

Visualization of the rendering loop is shown in Figure 3.30. There are two main branches of a computation: a red and a green branch. This branching is often called ping-pong calculations and it is used to remove dependencies of consecutive computations so they can be computed in parallel. As you can see, for computation of basis vectors  $b_c$  we need to know what coefficients  $\lambda_{ij}$  of probes looks like. The same goes for the computation of receivers irradiance, where we need to know basis vectors  $b_c$ . During run-time, CPU switches the red branch and the green branch in every draw call. This way every compute shader always has data ready from the previous cycle and the computation has to be blocked by a memory barrier only at the end of the draw call. However, this way there is a delay of four frames before newly come information will propagate all the way to the frame buffer.



**Figure 3.30.** Main rendering loop of the real-time application.

In Figure 3.30 you can also notice, that algorithm uses 3 different compute shaders and a single rendering pipeline. I will describe all the stages separately.

### 3.3.1.1 Computing SH Coefficients $\lambda_{ij}$ in Real-Time

The first compute shader computes SH coefficients  $\lambda_{ij}$  as it was shown in section 3.1.2 using Equation (3):

$$\lambda_{ij} = \int_{\vec{x} \in X} f(\vec{x}) Y_j(\vec{x}) dX$$

To calculate an integral in real-time from measured samples I used Monte-Carlo integration method. Using this method I can rewrite the equation as:

$$\lambda_{ij} \approx \frac{1}{N} \sum_{i=1}^N f(\vec{x}_i) Y_j(\vec{x}_i) w(\vec{x}_i) \quad (10)$$

where  $N$  is a number of samples (in our case a number of relight rays per probe),  $\vec{x}_i$  is a single relight ray,  $f(\vec{x}_i)$  is a measured irradiance at a hitpoint of ray  $\vec{x}_i$ ,  $Y_j(\vec{x}_i)$  is a value of basis function  $j$  in a direction of ray  $\vec{x}_i$ . This value was precomputed into **Spherical Harmonics texture** shown in section 3.2.5.  $w(\vec{x}_i)$  is the weight of a single sample. The weight of the sample is an inverted probability of the sample. As all rays have the same probability and we shoot rays over the whole sphere,  $w(\vec{x}_i)$  will be constant for every ray and equal to  $4\pi$ . So the final equation has the following shape:

$$\lambda_{ij} \approx \frac{4\pi}{N} \sum_{i=1}^N f(\vec{x}_i) Y_j(\vec{x}_i)$$

The result is stored in a texture, that will be used during cluster basis vectors computation.

### 3.3.1.2 Computing Cluster Basis Vectors $b_c$ in Real-Time

The second stage of the computation takes computed SH coefficients  $\lambda_{ij}$  and matrices stored in **Cluster Matrix texture** and compute cluster basis vectors  $b_c$ . To choose which probe coefficients multiply with which matrix, there is **Cluster Probe Index texture**. For a size of the cluster there is **Cluster Size texture**. Everything about data textures is in section 3.2.5. Computation itself just multiply the vector of SH coefficients with an appropriate matrix as it is shown in Figure 3.31.

$$\Sigma v^t \cdot \lambda_{ij} = b_c$$

**Figure 3.31.** Cluster basis vectors: matrix multiplication.

The result is stored in a texture to be used during receiver irradiance computation.

### 3.3.1.3 Computing Irradiance at Receivers in Real-Time

The last part of the computation writes to the lightmap texture. To obtain irradiance which should be written into the lightmap, each receiver has to multiply a basis vector of its cluster with his own transport vector. Matrices with transport vectors are stored in `Receiver Transport texture`. Computation equation for receiver  $x$  looks like this:

$$I(x) = \sum_{i=1}^S U_{x,i} b_{c,i}$$

where  $S$  is a number of used singular values after truncatization of SVD matrices (it is also a height of the matrix  $\Sigma V^t$ ),  $U_{x,i}$  is an  $i$ -th value in the transport vector for the receiver  $x$  and  $b_{c,i}$  is an  $i$ -th value in the basis vector of cluster  $c$  (cluster where  $x$  belongs).

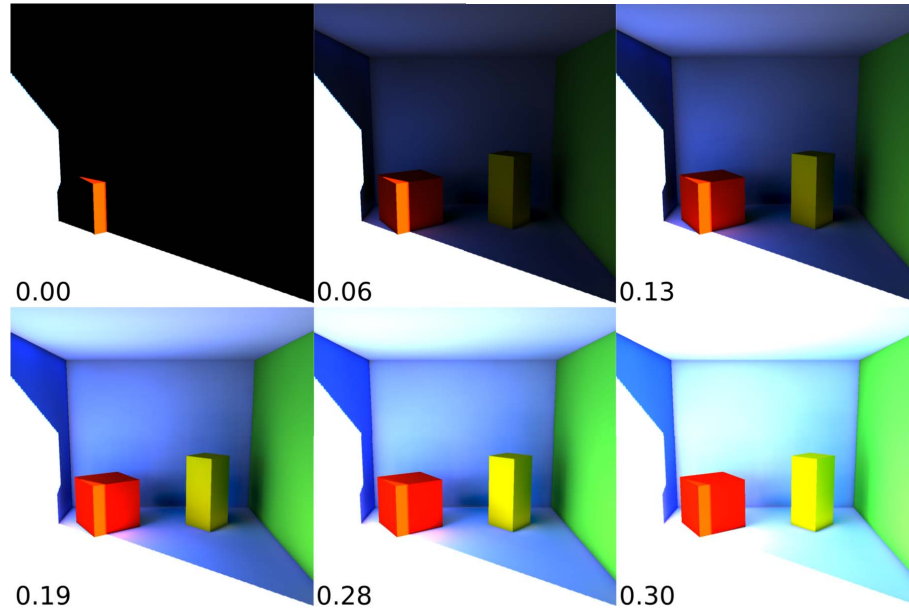
For the computation itself also other textures from section 3.2.5 are needed, but they are not important for an explanation of the algorithm.

### 3.3.1.4 Rendering Scene

During rendering, a standard rendering pipeline is followed. Every fragment find its irradiance in the lightmap texture. This means that every vertex of the mesh has two different UV coordinates: one UV coordinates for diffuse textures etc. and the second UV coordinates for the lightmap texture. After obtaining irradiance from the lightmap, tone mapping comes in place. In my work, I implemented three tone mappings: Reinhard, Filmic and Exposure tone mapping.

### 3.3.1.5 Multiple Bounce Irradiance Gathering

As it was mentioned in chapter 3.1.1, during the computation of  $\lambda_{ij}$  SH coefficients irradiance is collected even from the previously computed lightmap. This fact will cause reflection of light in the scene. Usage of the lightmap is even marked in Figure 3.30 as an arrow coming from an irradiance computation segment back to the  $\lambda_{ij}$  computation segment. Irradiance from previous cycle is added to  $f(\vec{x})$  in Equation (10). The amount of the reflected irradiance can be set either from a material reflection or as static reflection constant. In Figure 3.32 are few examples of a different static reflection constant setups.



**Figure 3.32.** Irradiance fetch-back examples. Different setups of a static reflection constant, from the top left to the bottom right: 0.0 , 0.06 , 0.13 , 0.19 , 0.28 , 0.3.

If the static reflection constant is 0, there is only a single bounce of light, from the light source to receivers. Higher static reflection constant means more bounces, but it takes system more frames to stabilize. Lower static reflection constant means fewer bounces, worse results, but a faster convergence of the system to a stable state. In my implementation, I set the static reflection constant to 0.19. If the static reflection constant would be bigger than 1.0, the system would receive every frame more and more irradiance and diverge to infinity.

### 3.4 Asymptotic Time and Memory Complexity

I chose to estimate the complexity due to a number of probes and a number of receivers, as these values are directly set up by the user. For both parts, I marked the number of probes with  $P$  and the number of receivers with  $R$ .

#### 3.4.1 Precomputation Algorithm Complexity

At the beginning, we must iterate over all texels of rendered textures to choose which texels create receivers:  $\Omega(R)$ . During the shifting phase, we iterate over all receivers:  $O(R)$ . Then we must obtain the initial set of probes from the grid. Thus, iterate over the whole grid:  $\Omega(P + N)$ , where the initial set has size  $P+N$ . From the initial set of probes, we must choose our final set. To do so, we calculate density function between all pairs from the initial set:  $O((P + N)^2)$ . Then we iteratively remove probes until we get the desired number of probes (which is  $P$ ):  $O((N - P)(P + N)) \sim O(N^2)$ . To calculate probe radii, we iterate over all receivers and for every receiver calculate a distance to every probe:  $O(RP)$ . Precomputation of probe hitpoints takes:  $O(Rk)$ , where  $k$  is a cost for shooting  $k$  rays. For each receiver, we obtain a set of the most influential probes:  $O(RP)$ . To measure coefficients for all receivers, we must again shoot rays:  $O(Rk)$ .

At the beginning of the clustering, we calculate the AABB of the scene:  $O(R)$ . In the worst-case clustering, we would get  $R$  clusters. Thus, doing  $2R + 1$  clustering operations. In every clustering operation, we find a unique set of probes:  $O(RP)$ , fill a transformation matrix  $A$ :  $O(RP)$  and compute Singular Value Decomposition:  $O(RP^2)$ . So the clustering gives us  $O((2R + 1)(RP + RP + RP^2)) \sim O(R^2P^2)$ . We must export positions for both probes and receivers:  $O(P + R)$  and transformation matrices for all clusters:  $O(R(P + R))$ .

Memory complexity of the precomputation part is mostly affected by the storage of coefficients for all receivers. Every receiver could possibly have stored coefficients for every probe:  $O(RP)$ . During clustering, memory demands cannot rise because of truncatization of the transport matrices.

In total, we get time complexity of the precomputation part  $T(R, P) \sim O(R^2P^2)$  with memory complexity of  $S(R, P) \sim O(RP)$ .

### 3.4.2 Real-time Algorithm Complexity

Real-time part of the algorithm is mostly computed in parallel on GPU, let the number of parallel computational devices be  $c$ . Firstly, we compute  $\lambda_{ij}$  SH coefficients. Thus, for every single coefficient we track precomputed hitpoints:  $O(3k)$ , where 3 stands for RGB. If this computation would appear on the single-threaded device, the complexity will be:  $O(P \cdot 64 \cdot 3k)$ , where 64 stands for 64 SH coefficients for a single probe. As the computation is done in parallel, we get:  $O(\frac{P \cdot 64}{c} \cdot 3k)$ .

To obtain a cluster basis vector, we multiply  $\Sigma V^T$  with a vector full of  $\lambda_{ij}$  SH coefficients. Matrix  $\Sigma V^T$  height can be at most 32 (as we truncated SVD matrices to 32 most influential singular values), a width of the matrix can be at most  $P$  (all probes support this cluster). Thus, to obtain a single value from the cluster basis vector, we multiply two vectors with a length of  $P$ :  $O(P^3)$ , where, 3 stands for RGB. To obtain the whole cluster basis vector, we must do this 32 times:  $O(32 \cdot P^3)$ . In the worst-case clustering, we get  $R$  clusters with a single receiver which is supported by all probes. In this case, time complexity is:  $O(RP^3)$ . If we do this operation in parallel, we get:  $O(\frac{R}{c} P^3)$ .

Irradiance computation consists of multiplication of two vectors with the length of 32 numbers. Thus, a single irradiance value can be computed in:  $O(32 \cdot 3)$ , where 3 stands for RGB. In total for all receivers we get:  $O(R \cdot 32 \cdot 3)$ . If we use parallel computation, the problem can be solved in:  $O(\frac{R}{c} \cdot 32 \cdot 3)$ .

Memory complexity of the real-time part is dependent solely on the transformation matrices. A number of all  $\lambda_{ij}$  SH coefficients is  $O(P \cdot 64)$ . In the worst-case clustering, we get  $R$  clusters with a single receiver. So, there will be  $R$  times  $\Sigma V^T$  matrix with single row and  $P$  columns.

The transformation matrix  $U$  would have a height and a width of 1.

In total, we get time complexity of the real-time part  $T(R, P) \sim O(\frac{RP}{c})$  with memory complexity of  $S(R, P) \sim O(RP)$ .



## **Chapter 4**

### **Method Extensions**

In my work, I implemented two extensions with an aim to push this method towards usage in real-time rendering engines. My first extension is capable of shading dynamic objects, so they can receive indirect illumination from a scene. This extension can do much more. All the capabilities of this method are described in section 4.1.

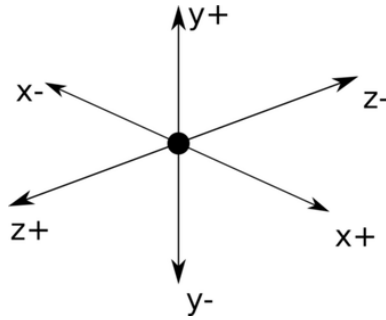
The second extension is dynamic Level of Detail (LOD), where only those irradiance receivers which are necessary to render the correct frame are computed. This method can improve the performance on every stage of computation. Again, more is written in section 4.2.



## 4.1 Shading Dynamic Objects Using Voxel Irradiance Space

To gather irradiance for dynamic objects I combined the idea of spatial irradiance caching [22] with the main implemented method [4]. However, as the work of Greger et al. [22] was demanding on memory and computation, I used only the idea of measuring irradiance in the grid. My approach to irradiance storage and irradiance gathering is different and saves both memory and computation time.

To measure irradiance on a surface of dynamic objects I used a special version of irradiance receivers, I call them spatial receivers. These receivers measure irradiance from certain directions, particularly in such directions to be able to estimate irradiance in all directions. In my implementation, a single spatial receiver is measuring irradiance from 6 different directions as is shown in Figure 4.1. However, the method is not constrained to a fixed number of directions. More directions can provide a better approximation of irradiance.



**Figure 4.1.** Visualization of a spatial receiver using 6 directions to measure irradiance in a space.

In every direction, I measure irradiance with a weight equal to a normalized degree between the direction of incoming irradiance and a current direction vector. I pre-compute Spherical Harmonics weighted functions for every direction. The weight is computed with the following equation:

$$w(d, I) = \frac{\frac{\pi}{2} - \text{acos}(\max(\text{dot}(d, I), 0))}{\frac{\pi}{2}}$$

This equation gives us an ability to combine irradiance from different directions of the spatial receiver into a single value. More about irradiance reconstruction will come later.

When clustering spatial receivers, I chose to cluster every direction separately as it was more coherent. The biggest benefit obtained from clustering spatial receivers separately from standard receivers comes from GPU computation itself. In my implementation, I have a single compute shader which computes both receivers and spatial receivers (as a lot of the code is the same). In the shader, one of the first thing which is decided is if the currently computed receiver is a spatial or a standard one. So, there is a branch which leads to big chunks of code. This would be a problem if receivers and spatial receivers were mixed together. If they would be assigned in the same warp, both branches would have to be executed. Because they are clustered separately, all receivers come always before spatial receivers. As a result, there is either one or no warp which has to go through both branches of the computation. Now I will describe how I placed spatial receivers in the scene.

### 4.1.1 Irradiance Voxel Space Using Regular Grid

To be able to reconstruct irradiance in every point of scene I voxelized space with a regular grid, where every vertex of the grid is a single spatial receiver. As input from the user, the algorithm takes an approximate number of desired spatial receivers. Using the following equations (1), (2), (3) (from the work of Amanatides and Woo[29]) I estimate the number of spatial receivers in every axis.

$$N_z = \left\lceil \sqrt[3]{\frac{Nz^2}{xy}} \right\rceil \quad (1)$$

$$N_y = \left\lceil \sqrt{\frac{Ny}{N_zx}} \right\rceil \quad (2)$$

$$N_x = \left\lceil \frac{N}{N_yN_z} \right\rceil \quad (3)$$

$N_z, N_y, N_x$  is a number of cells in z-, y-, x-axis,  $N$  is a desired number of cells and  $[x, y, z]$  is a size of the scene's AABB.

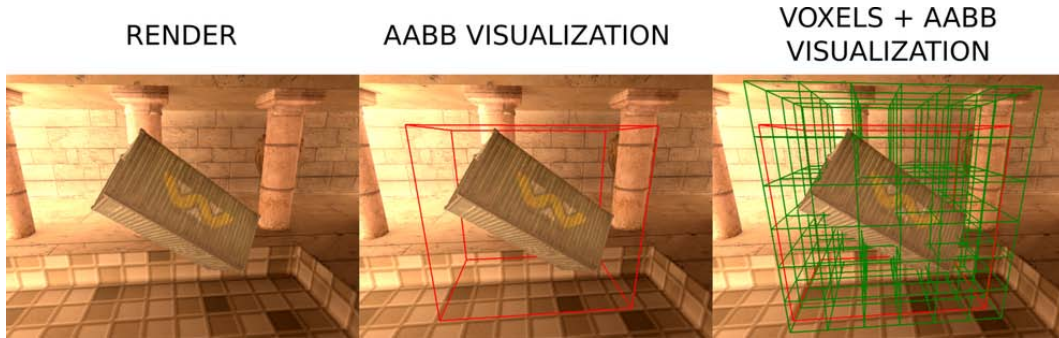
Important note: it can happen, that some spatial receivers will be placed inside objects of the static scene. These receivers would not receive any irradiance. This would distort the resulting grid. To avoid this artifact, I used the same approach as for standard receivers (see section 5.1.4). I shoot testing rays around the spatial receiver. Here, however, I chose a different rule. If 80% of rays hit backface (no matter rays which flew out of the scene) then the spatial receiver is shifted. Why omit rays which flew out? The scene grid does not have to cover the scene completely. It can happen that some spatial receivers are out of the scene because of the scene shape. If I left the old rule (do not shift if any ray flew out) I would not shift any spatial receiver which is placed out of the scene.

Choosing a regular grid as storage for spatial receivers got several reasons. The first, the most important one, is the capability of HW linear interpolation using OpenGL 3D textures. This way I avoided searching nearest spatial receivers for every fragment of dynamic objects. Using interpolation I can use a relatively sparse grid to reconstruct shading for dynamic objects.

For the second, it is possible to set extension of borders to infinity on 3D textures in OpenGL. This allows to approximate irradiance even in points which are not directly in the grid. So, the grid does not have to cover the whole scene.

For the third, it is easy and cheap to find correct coordinates to a regular 3D texture. And for the fourth, I can easily detect which parts of the grid have to be computed and which can be omitted.

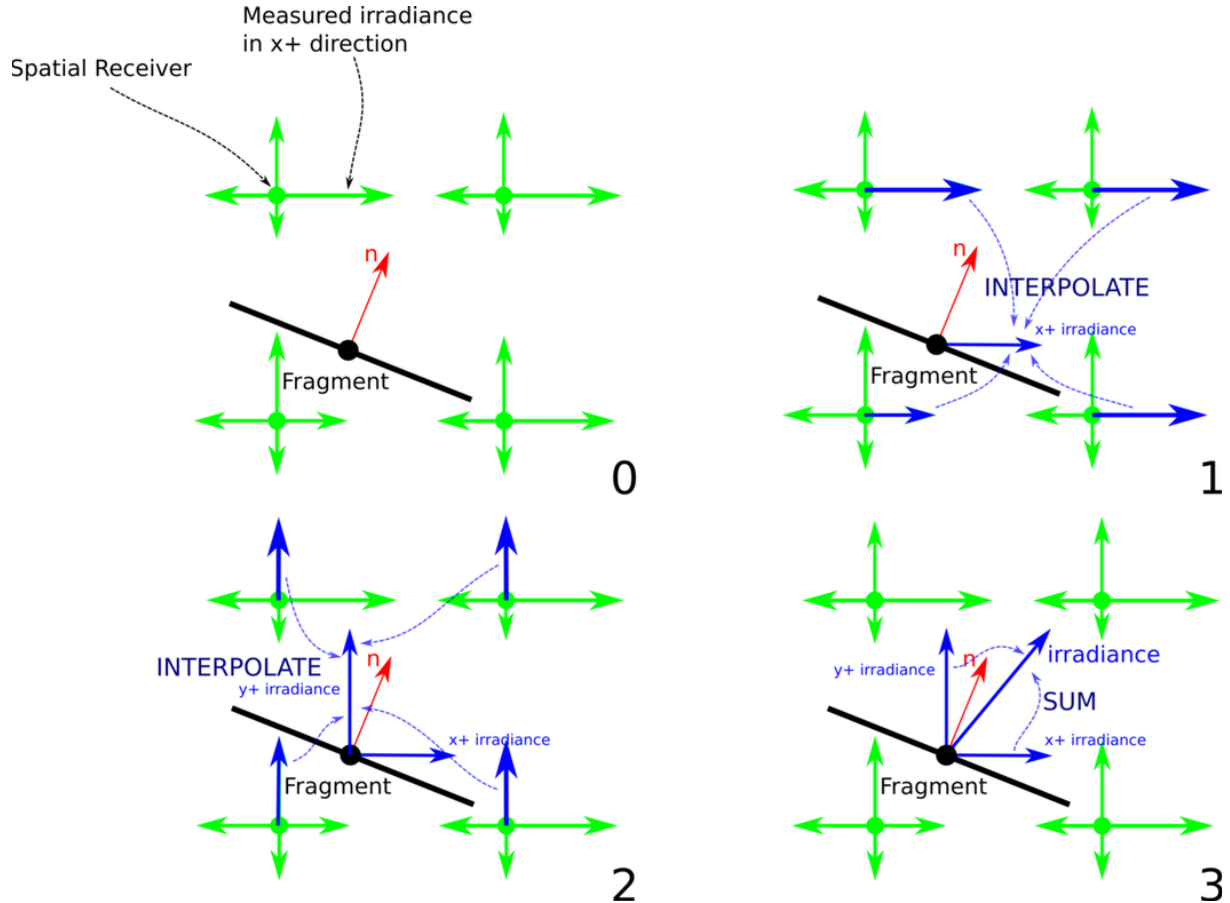
To extend this idea, in my implementation every dynamic object has its AABB. This AABB is tested against the irradiance grid, so I can detect which spatial receivers have to be computed. Visualization of this dynamic computation is shown in Figure 4.2. Active voxels are drawn with green color, the AABB of a dynamic object (model of cargo container) is red.



**Figure 4.2.** Visualization of active spatial receivers (green voxels) triggered by dynamic model of cargo container (red AABB).

### 4.1.2 Computation of Irradiance from Voxel Grid

For dynamic objects, computation of irradiance is done per fragment. In a fragment shader, every fragment determines its position in the 3D irradiance texture. Because a normal vector of the fragment is known, it is possible to estimate which directions of spatial receivers would affect this fragment the most. I measure weight for every spatial receiver direction and summarize them with an irradiance together to gather final irradiance contribution. Visualization is shown in Figure 4.3.



**Figure 4.3.** Visualization of irradiance reconstruction in a fragment. A spatial receivers (green) in a grid, the fragment (black) with a normal vector (red) and interpolated irradiances (blue). Summed irradiance is marked with **irradiance** label (light vector).

The light vector describes from which direction comes the maximal valid irradiance to a certain fragment. Notice how this method gives us an approximation of the light vector even though irradiance was transferred. This allows using normal mapping and specular reflections for dynamics objects. Furthermore, if we approximate these light vectors even for fragments which are part of a static scene, we can apply normal mapping and specular reflections to the static scene as well. This approximation is not as accurate as transferring radiance vectors, but it cost almost no additional performance and results are visually pleasing. For results using this method see chapter 5.

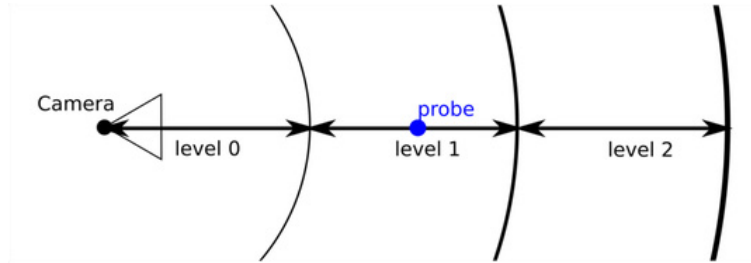
## 4.2 Dynamic Level of Detail

Computation of light in my implementation is separated to 3 separate stages: computation of SH coefficients for probes ( $\lambda_{ij}$ ), computation of cluster basis vectors and computation of irradiance for every receiver using cluster basis vectors. My method is affecting every stage of this computation, so I will describe them sequentially. Every aspect of LOD, such as distances, number of coefficients, a number of relight rays and so on can be freely changed in runtime.

For every level of detail, I define a distance, where this level is triggered. The distance is measured between the object and a center of the camera and then compared with defined LOD distances.

### 4.2.1 Probe Computation with LOD

Firstly I determine to which LOD certain probe belongs. I measure this via distance between the camera center and a probe position in space as is shown in Figure 4.4.



**Figure 4.4.** Visualization of LOD distance for probes.

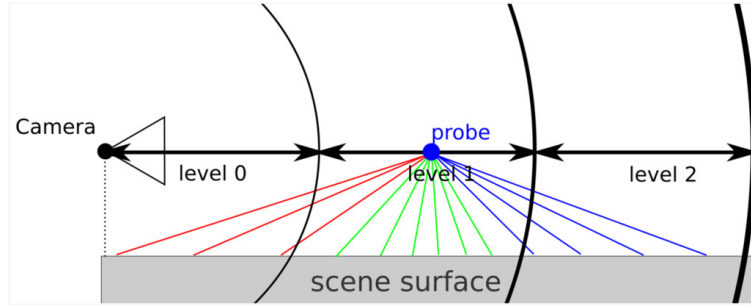
With a known level of detail for a certain probe, it is possible to reduce calculation time in many ways.

The first place where I reduced calculation time is in the number of coefficients which are computed. To explain I'll remind a little bit of theory about Spherical Harmonics. As it was said in section 3.1.2, Spherical Harmonics can be seen as a general function  $f(\omega)$ . If we put some unit vector (direction in 3D) to this function, the function will return value (in our case direction  $\omega$  is the direction of light and the returned value is irradiance). To create this function  $f(\omega)$ , Spherical Harmonics use basis functions. With the combination of these functions, it is possible to reconstruct (with some deviation) a signal encoded to the function. This is similar to Furrier transform functions. Combinations of SH basis functions are encoded in so-called SH coefficients ( $\lambda_{ij}$ ). The more coefficients (and SH basis functions) we use, the more accurate will be a reconstruction of the signal. So let us say we have SH of degree 4 (which is 25 coefficients, 25 basis functions to reconstruct function  $f(\omega)$ ), now we use SH of degree 7 (which is 64 coefficients). But even if we use SH of degree 7, those 25 coefficients from 64 in total will affect exactly the same basis functions as before.

I used this property in LOD, so I calculate only that number of coefficients for each probe, which is necessary for correct irradiance reconstruction. For example, in level 0 I use 64 coefficients, in level 1 25 coefficients, in level 2 only 16 coefficients and so on. Of course, this number of coefficients must match with the computation of cluster basis vectors, but I will get to that later.

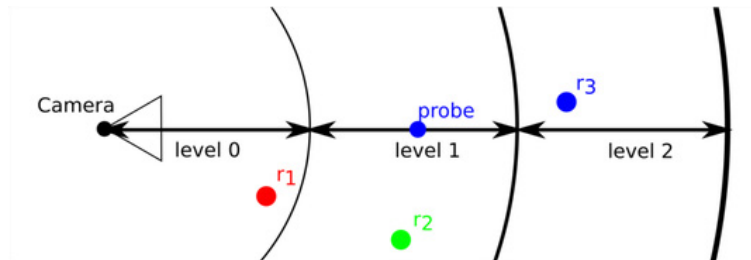
The second major improvement was made in the number of relight rays. Relight rays are used to approximate  $\lambda_{ij}$  coefficients using Monte-Carlo integration. The more relight rays you use, the more accurate SH coefficients are, so the reconstruction of irradiance is more accurate. To save computation time it is desirable to lower the amount of these rays. In my implementation, I used 512 for level 0, 256 for level 1, 128 for level 2 and so on.

With these extensions, you can notice some problems. The first problem is that a relight ray hit does not have to be in the same LOD as the probe is. This problem is visualized in Figure 4.5.



**Figure 4.5.** Visualization of different LOD for a probe and its relight rays, red rays require more detail, green rays are at same LOD and blue need less detail.

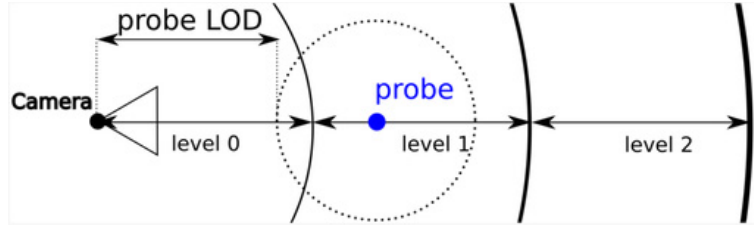
To solve this problem, every relight ray needs to look into the lightmap texture to correct LOD. There are two possible solutions on how to solve this. The first solution is, that every relight ray remembers its 3D position of a hit. The relight ray then decides itself to which LOD to look for the irradiance. The second solution is that relight rays always take irradiance from the lightmap texture with highest LOD, because this LOD is refreshed every frame and contains the newest information (more about the lightmap texture computation in section 4.2.3). The second solution provides faster but less accurate reconstruction of the signal. In my implementation, I made both solutions. Another problem comes from receivers, see Figure 4.6.



**Figure 4.6.** Visualization of different LOD for a probe and receivers supported by this probe.

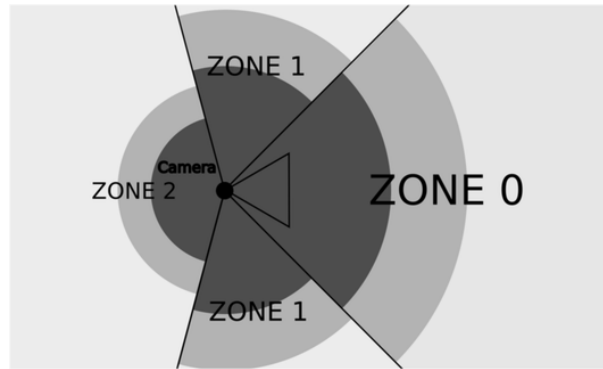
Even though the probe is in level 1, we cannot assign him to level 1 because there is receiver  $r_1$  which is in level 0 and needs more detail.

However, when the scene was precomputed, only receivers which fell under the support of a certain probe were affected by it. Support of every probe was determined by a constant radius. Thus, the solution is to shift LOD of the probe by the constant radius to make sure that every receiver which needs more detailed information will get it. Visualization of this shifting is shown in Figure 4.7.



**Figure 4.7.** Visualization of different probe LOD shifted by the probe radius.

To further improve performance I changed LOD for every probe (and its relight rays) with respect to the camera point of view. I split surrounding of the camera into separated zones as is shown in Figure 4.8. To maintain good performance I implemented this separation using dot product with the camera view direction. This technique has been used for cluster basis vectors and irradiance receivers as well.



**Figure 4.8.** Visualization of separated zones around the camera, each zone has different LOD distances.

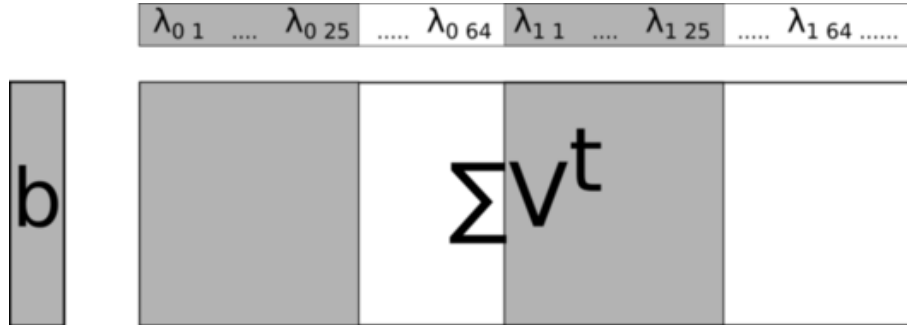
It is important to calculate even points which are not directly visible to the viewer. Due to global illumination, even not visible points affect the final image.

### 4.2.2 Cluster Basis Computation with LOD

Firstly, we have to determine to which LOD certain cluster belongs. In the precomputation part, clusters were made by an kD-tree division. So, every cluster fits nicely into AABB. From the precomputation I saved this AABB (I refitted the AABB to reduce free space) then I measured the closest distance from the camera center to this BV and use the camera view dependant zones as in section 4.2.1.

With determined LOD of the cluster, I can determine how many coefficients from probes to use for basis vector. A number of coefficients for every LOD is constant through the application. In the precomputation part, two PCA matrices were calculated: matrix  $\Sigma V^t$  and matrix  $U$ . In this stage, we compute a basis vector  $b$  so now we only need matrix  $\Sigma V^t$ .

An important property of SVD is that it will not change the order of input transformation vector. Thus, the order of columns in matrix  $\Sigma V^t$  is equal to the order of columns in the original transformation matrix, let's call that matrix  $A$ . So, if the input vector is some vector  $x$ , it means that  $Ax = U\Sigma V^t x$ . In the algorithm, our input vector is a vector containing SH coefficients  $\lambda_{ij}$  of probes supporting this certain cluster. Because columns are in the correct order even after decomposition, we can use only those columns from  $\Sigma V^t$  which we need. For example, let us say we want to use only 25 coefficients, but our PCA matrices are computed for 64 coefficients. We will match appropriate columns with input SH coefficients  $\lambda_{ij}$  to compute the basis vector only up to those 25 coefficients, visualization is in Figure 4.9.



**Figure 4.9.** Visualization of partial matrix multiplication to obtain cluster basis vector  $b$ . Grey parts are used/computed, white parts omitted.

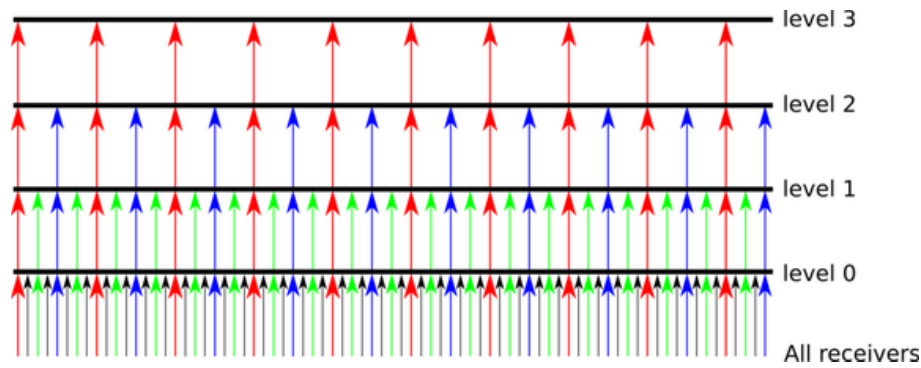
LOD of each cluster is stored into a texture and used in receiver computation.



### 4.2.3 Irradiance Receiver Computation with LOD

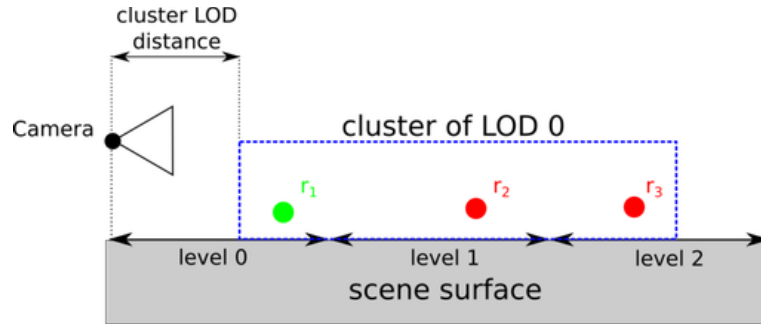
In this section of computation, irradiance is computed for every receiver, so even for spatial receivers. Before the irradiance is computed, a receiver is tested if it is necessary to calculate it. The test is different if the receiver is a standard texel of lightmap texture or if it is a spatial receiver.

If it is a standard receiver, the test starts with reading LOD of its cluster from a texture. Then, the level of this receiver needs to be known. For obtaining the level of receiver we follow the same rules as for standard mipmapping. Let's say that coordinates of the receiver in the lightmap texture (in pixels) are  $(x, y)$ . If for both coordinates the last bit is 0, then this receiver is at least level 1. If 2 last bits are 0, then this receiver is at least level 2 and so on. Otherwise, the receiver is level 0. For example, if the cluster of this receiver is in LOD 1, every receiver with level bigger or equal than the level of its cluster will be computed (every receiver with level 1,2,3... but not 0). Level of the receiver also decides to how many LODs of the lightmap texture will the irradiance value be written (see Figure 4.10). Let us say we have a cluster in LOD 2. Only receivers with level 2 and 3 will be computed (and propagated to LODs 2 and 3).



**Figure 4.10.** Visualization of receivers writing its irradiance through LODs, receivers of level 0 got black color, receivers of level 1 are green, level 2 is blue, and level 3 is a red color.

Important is that we cannot write irradiance only to that LOD which cluster has, but we always have to propagate the resulting irradiance to the level of a certain receiver. A reason for this is that even though cluster can be for example in LOD 0 (the closest point from a cluster AABB was in LOD 0) not every receiver has to be in this LOD. So, we may require lesser detail during rendering from the cluster with higher LOD, as it is visualized in Figure 4.11.



**Figure 4.11.** Visualized problem with receivers being in different LOD than their cluster, receiver  $r_2$  and  $r_3$  are in LOD 1 and 2 respectively, but the cluster is in LOD 0.

The second case is if the receiver is a spatial receiver. In this test, the spatial receiver is checked if it is in a list of active receivers needed for the computation (if some dynamic object triggered a voxel with this receiver as its vertex). The test is visualized in Figure 4.2.

#### 4.2.4 Usage of LOD Irradiance Textures in Fragments

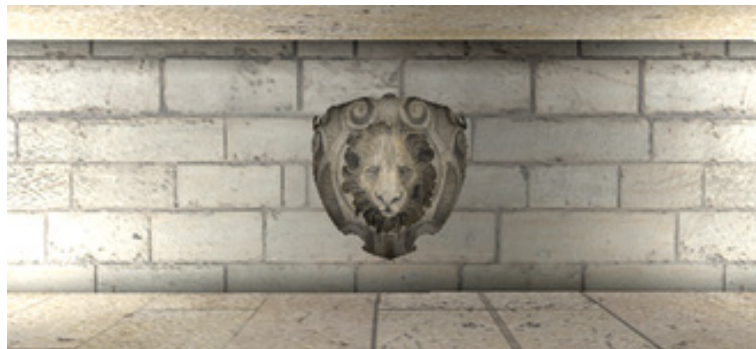
LOD for every fragment of a static scene is determined exactly the same way as it was shown in Figure 4.4. In a naive solution, the fragment just looks up its irradiance in the appropriate lightmap texture LOD. This solution brings two major problems. The first problem is an obvious one: there is a sharp edge between fragments in different LODs. An example of this is shown in Figure 4.12.



**Figure 4.12.** Fragments take irradiance without smoothing between LOD, visible circle artifact around the lions head.

To solve this artifact, I smooth irradiance in fragments between LODs of the lightmap texture with a constant width of a smoothing range. A blending of irradiance always happens from more detailed LOD to less detailed LOD, never the other way around. If some LOD  $i$  is computed then necessarily LOD  $i+1$  is computed as well, but this does not have to be the case for LOD  $i-1$ .

A result of this improvement is shown in Figure 4.13.



**Figure 4.13.** Fragments take irradiance with smoothing between LOD, artifacts disappeared.

The second problem with fetching texels from the lightmap texture happens when the system still balances change of light in the scene. Thus, when some fragment takes irradiance from part of the system, which has not to converge yet, it starts flickering. This happens because the borders of LODs for the computation part are the same as for the rendering part.

The problem can be solved simply by shifting LOD distance for fragments. Not the best-computed irradiance texture is shown using this solution, but there is a high possibility that the system already converged in that part of the scene. Convergence of the system is highly dependent on a difference of irradiance between computed and not computed parts of the scene, a speed of camera movement through the scene and a refresh rate of irradiance computation.

## Chapter 5

### Results

To test both the proposed method [4] and my extensions, I implemented an application using C++ 14 and OpenGL 4.3. The application has parameters settable in run-time using Nuklear GUI<sup>1</sup>. The rest of the parameters is then settable using a configuration file. A complete guide on how to use the provided application is located in appendix A. Screenshot from the application is shown in Figure 5.1.



**Figure 5.1.** Screenshot from the implemented application.

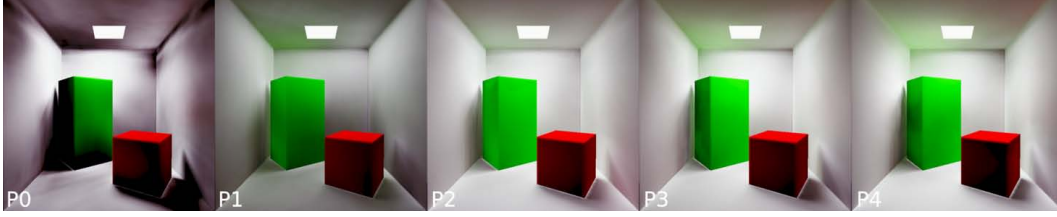
<sup>1</sup> <https://github.com/vurtun/nuklear>

## 5.1 Real-Time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes

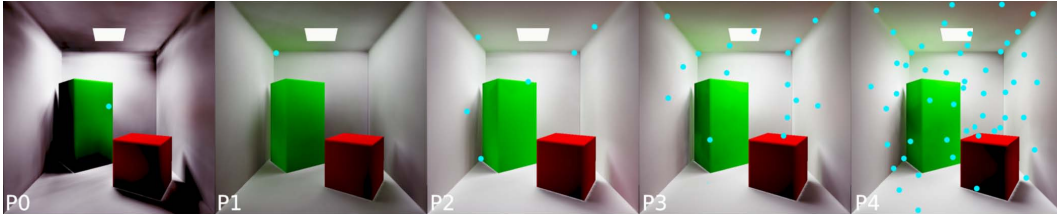
In this chapter, I am going to test several parameters of the proposed method [4] which are configurable. Testing has been done on laptop Lenovo G580: Intel(R) Core i5-3210M CPU @ 2.50GHz, RAM: 8GB, GPU NVIDIA 630m 1GB, 800 Mhz.

### 5.1.1 Testing Number of Probes

A number of probes crucially affects the quality of the final render. The more probes we use, the more accurate the signal reconstruction will be. However, computation of probes during run-time is costly, due to integration over hundreds of relighting rays. I tested this dependency as it is shown in Figure 5.2. In Figure 5.3 are visualized positions of probes in the scene as blue dots. You can notice how a single probe in image 0 is not sufficient for light reconstruction. However, 11 probes in image 2 provide a similarly good result as 78 probes in image 4.



**Figure 5.2.** Testing renders for a different number of probes, from the left to the right, from the lowest to the highest.

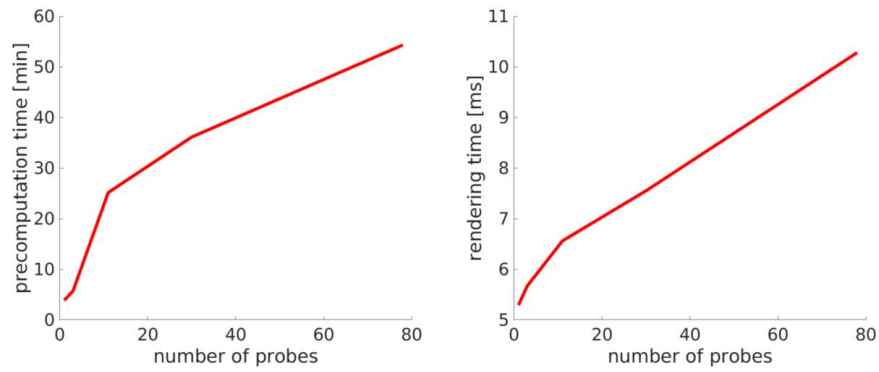


**Figure 5.3.** Testing scenes for a different number of probes with visualized probes position.

In Table 5.1 are setups used for each render. Figure 5.4 shows how the number of probes affects computation time. 65 011 receivers were used.

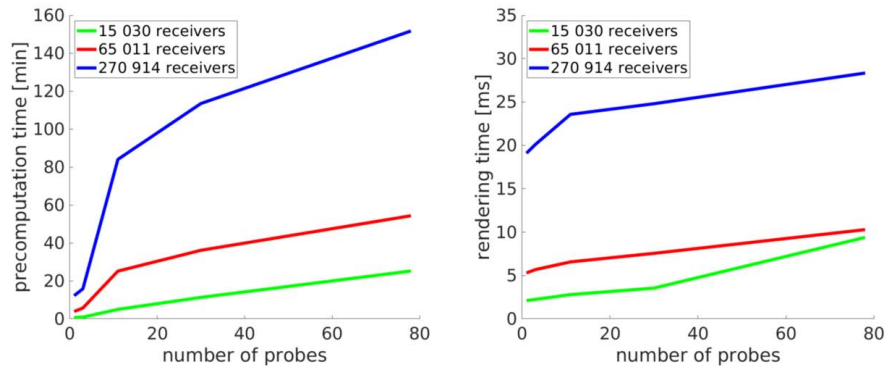
image	P0	P1	P2	P3	P4
precomputation time [min]	3.84	5.70	25.14	36.12	54.31
render time [ms]	5.29	5.67	6.56	7.55	10.28
number of probes	1	3	11	30	78

**Table 5.1.** Setups for testing number of probes in the scene.



**Figure 5.4.** Testing scenes for a different number of probes, on the left is a graph for the precomputation, on the right is a graph for the rendering.

In Figure 5.5 is comparison of times for a different number of receivers.



**Figure 5.5.** Testing scenes for a different number of probes and a different number of receivers. On the left is a graph for the precomputation, on the right is a graph for the rendering.

### 5.1.2 Testing Number of Receivers

For accurate lightmap calculation, a dense set of receivers is needed. However, the more receivers we have, the longer the computation takes. As a demonstration, I rendered the scene with several setups. Results are shown in Figure 5.6. For every tested setup there were 7 probes in the scene. Notice, how edges of meshes have blended irradiance from neighboring faces in low-resolution lightmaps. This is caused by linear filtering together with mesh unwrapped UV coordinates. The artifact disappears with higher resolutions.



**Figure 5.6.** Testing renders for a different number of receivers, from the left to the right, from the lowest to the highest.

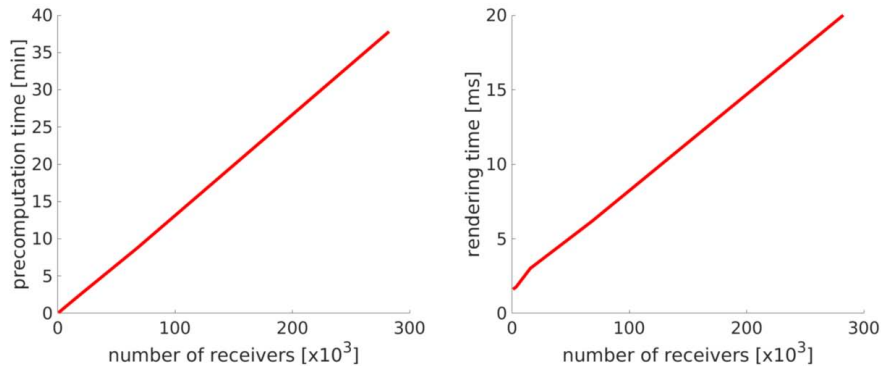


In Table 5.2 is information about all rendered scenes from Figure 5.6.

image	R0	R1	R2	R3	R4
precomputation [min]	0.09	0.45	2.06	8.75	37.8
render [ms]	1.61	1.76	3.02	6.15	19.19
receivers	712	3 369	15 608	67 662	282 403
clusters	2	4	36	103	428
lightmap size	$64 \times 64$	$128 \times 128$	$256 \times 256$	$512 \times 512$	$1024 \times 1024$

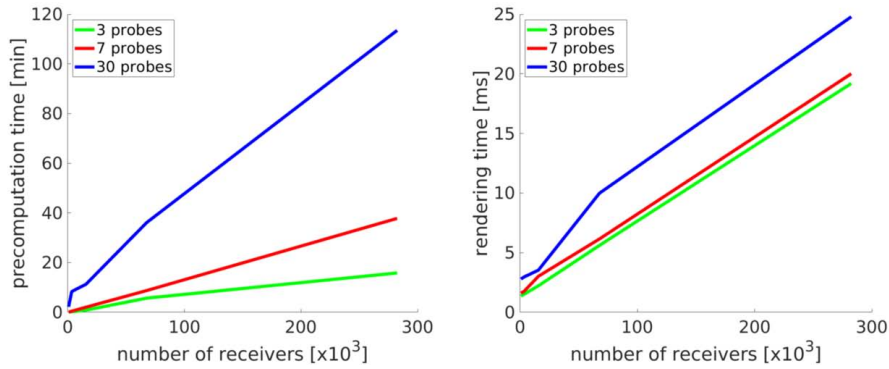
**Table 5.2.** Setups for testing number of receivers in the scene.

Figure 5.7 visualizes data from table 5.2 in graphs. Notice how both graphs are pretty much linear. This shows how impactful is the number of receivers on computation time.



**Figure 5.7.** Testing renders for a different number of receivers, on the left is a graph of precomputation part, on the right is a graph of rendering part.

Figure 5.8 shows a comparison of a different number of probes.



**Figure 5.8.** Testing renders for a different number of receivers, multiple setups. On the left is a graph of the precomputation part, on the right is a graph of the rendering part.

### 5.1.3 Testing $\alpha_{ij}$ Coefficients Setup

The proposed method [4] stored weighted visibility for irradiance receivers in  $\alpha_{ij}$  coefficients.  $\alpha_{ij}$  coefficients had several problematic aspects I had to solve. This chapter describes how I approached these problems in my implementation.

#### 5.1.3.1 No Hitpoint During Ray-Casting

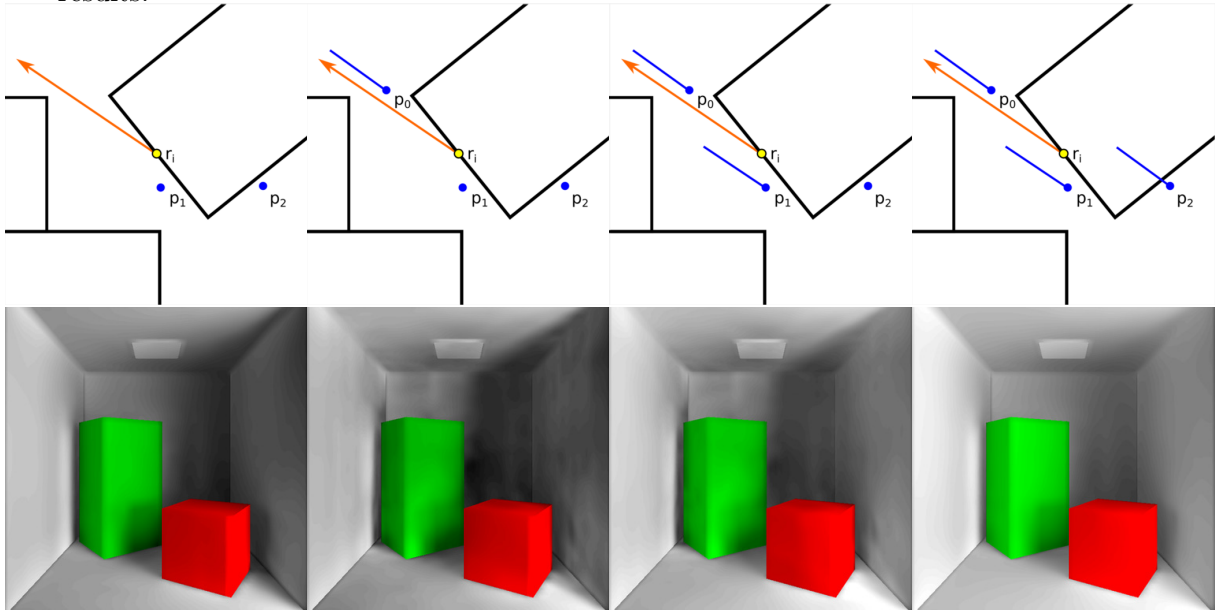
There is an undefined solution to the state where a ray hits no object during measurement of  $\alpha_{ij}$  coefficients. During testing, I created four separate solutions, which I will now describe. You can see a visualization of solutions together with appropriate renders in Figure 5.9. Renders have enhanced contrast for better light leaks visibility.

In the first column is an initial solution. I add zero contribution to the receiver if the ray had no hitpoint, the solution is in the first column, second row.

In the second solution (the second column) I added a contribution from probes that in the same direction as the receiver see nothing as well. You can notice visible light leaks due to unequal transfer function on the surface of meshes.

In the third solution (the third column) I added contribution only from probes which directly see the receiver.

The fourth solution is in the fourth column. Here I added an equal contribution from all probes no-matter visibility. This solution does not suffer from light leaks, but notice incorrect irradiance on the left face of the red cube. In comparison to other solutions, you can notice irradiance should be much weaker than it is. Thus, this solution is incorrect. For my final implementation, I chose the first solution as it provided the best results.

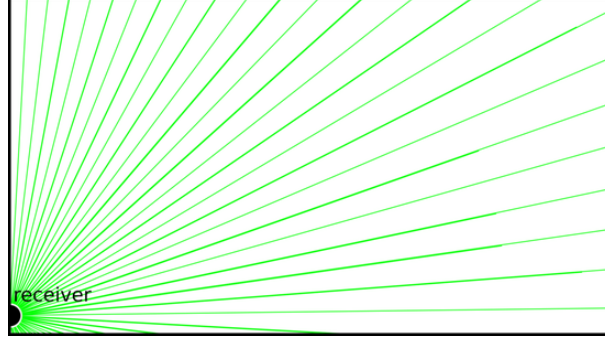


**Figure 5.9.** Receivers, no hitpoint example. From the left to the right: zero contribution, visible light leaks; the same direction seen by probes; contribution from probes visible to the receiver; contribution from all supporting probes.



### 5.1.3.2 Weighted Function for Ray Hitpoints

When casting a limited number of rays uniformly distributed across the hemisphere, surfaces which are closer to ray-casting starting point will have a higher sampling density of rays which hit them. This affects final measured coefficient  $\alpha_{ij}$  in a negative way. Coefficients which have been measured for receivers in corners will tend to have strong deviation towards closer surfaces. An example of this problem is shown in Figure 5.10, where a receiver is visualized as a black dot and measuring rays are green. Notice how the bottom part of the screen has a lot denser hitpoints then, for example, the ceiling.

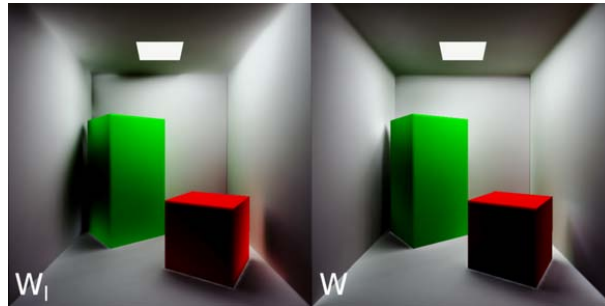


**Figure 5.10.** Testing weighted function for ray hitpoints, visualization of the problem.

To solve this problem, I weighted samples due to their hitpoint distance from a starting point of ray-casting. As the rays are cast from a point, we can imagine their probability of appearance at a certain distance as a projection on a surface of a sphere (hemisphere in this case). The probability of a ray appearing in a distance  $x$  decreases the same way as the surface of the hemisphere rises. I used this fact to weight all rays equally. The weight is then equal to the inversed normalized surface of a hemisphere:

$$W(x) = 1 - \frac{1}{2\pi x^2 + 1}$$

To support my statement, I compared this weight function with the previous, linear weight function  $W_l(x) = x$ . Results are shown in Figure 5.11, where the left picture is rendered with the linear weight function  $W_l(x)$  and the right picture with the new weight function  $W(x)$ . Notice, how corners and tight spaces have a tendency to be darker on the left picture.



**Figure 5.11.** Testing weighted function for ray hitpoints: image  $W_l$  was rendered with linear weight function  $W_l$ , image  $W$  was rendered with  $W$  weight function.

#### 5.1.4 Incorrect Irradiance at Contact Points of Meshes

You may notice in renders that bright aura where different objects are in contact. For example in Figure 5.12, where the red cube touches the floor.



**Figure 5.12.** Incorrect irradiance at contact points of meshes, example of a bright aura.

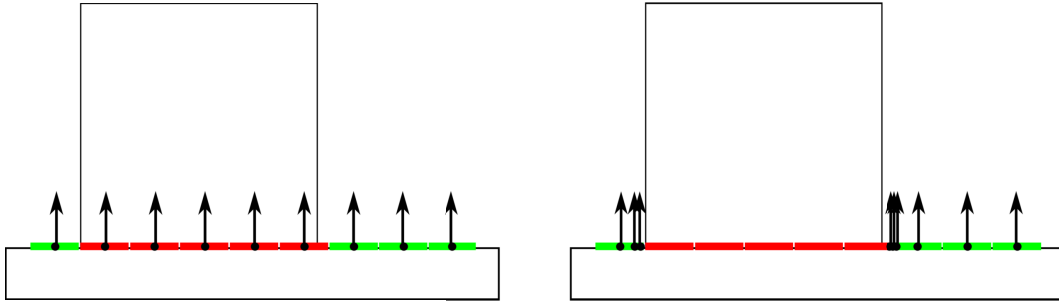
This aura is mainly caused by linear interpolation of texture. The aura will be smaller if receivers are dense enough as it is shown in Figure 5.13.



**Figure 5.13.** Incorrect irradiance at contact points of meshes, how higher sampling dims the aura.

Another reason which causes the problem comes from the used ray-caster. The ray-caster I used had backface-culling enabled. Thus, if a starting point of a ray is too close to another object, the ray will fly into that object. Also, because of backface-culling, the ray will go through the object and fly out the other side. This false ray will then measure irradiance at some distant part of the scene. Note that even with backface-culling turned off, incorrect irradiance would still occur, but there will not be a bright aura but a dark one. The dark aura would be less visible in more conditions, but the results would be false nonetheless.

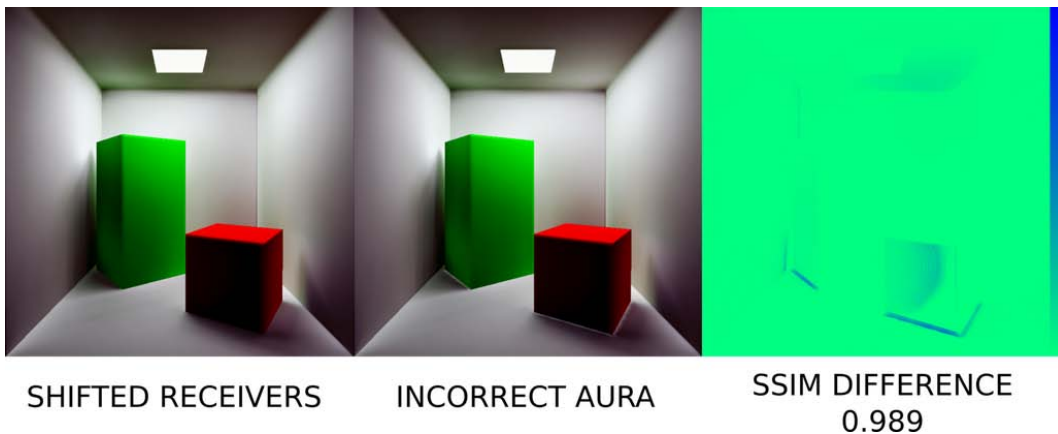
To solve this problem or at least dim the aura effect as much as possible, I included a new part of the algorithm to the precomputation of the scene. When receivers are created, their position in space is saved. From this position, their SH coefficients  $\alpha_{ij}$  are then computed. My idea is to shift their position used for SH coefficients calculation. For better imagination, my solution to this problem is shown in Figure 5.14. Colored rectangles represent texels of the lightmap texture, green texels are correct, red texels are wrong. Dots and arrows represent from which point certain receiver measured its SH coefficients.



**Figure 5.14.** Incorrect irradiance at contact points of meshes, shifting receivers position. On the left is the original position of receivers, on the right is shifted position of incorrect receivers.

Determining where to shift a receiver is not trivial. Surfaces and contact places of meshes can have a huge variety of these problematic spots. To solve this problem for general cases I chose a rather slow but reliable technique. From every receiver, I shoot testing rays. If a testing ray  $r$  hits backface of any mesh, I measure a length of this ray and choose the shortest ray from all rays which hit backface. If at least 60% of rays hit backface and no ray flew out of the scene, I accept measured shifting. And finally, I shift the position of the receiver in a direction (and distance) of this shortest ray. An offset is involved to be sure that the shifted receiver is outside of the mesh. I store this shift by overwriting receiver's position in 3D with the shifted position. Its texel position in lightmap never changes.

Resulting render using this technique as well as old render with the aura and SSIM comparison is shown in Figure 5.15.



**Figure 5.15.** Comparison of render with shifted receivers position and the original render without shifting. The last image is an SSIM comparison.

Notice how different the final render is when receivers were shifted. This is a good example of how subtle changes completely change the final image when global illumination techniques are used.

## 5.2 Extensions

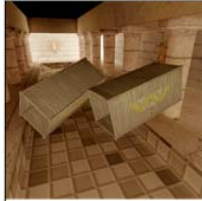
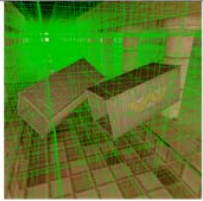
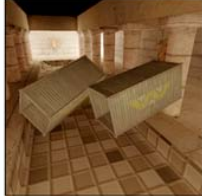
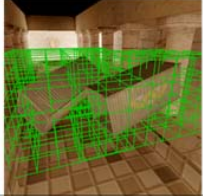
I implemented two major extensions for the proposed algorithm [4]. Testing and results of these extensions is this chapter. Please visit the following URL<sup>1</sup> for accompanying video.

### 5.2.1 Global Illumination Shading for Dynamic Objects

Testing of my first extension capable of shading dynamic objects using indirect illumination is in this chapter.

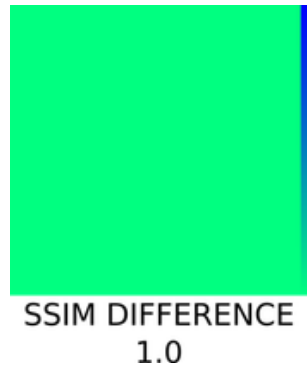
#### 5.2.1.1 Testing Voxel Grid - Dynamic Voxel Irradiance Calculation

For testing purpose, I precomputed a scene with a voxel grid of size  $[26 \times 9 \times 64]$ . So, there are 14 976 spatial receivers in total. I placed two models into the scene. Then I measured the time and the number of active voxels during computation. The resulting renders with an active voxels visualization are visible in Figure 5.16. Achieved speedup of the computation was 1.85.

	RENDER	GRID VISUALIZATION	COMPUTED RECEIVERS	TIME [ms]	SPEEDUP
ALL SPATIAL RECEIVERS			14 976	14.42	1.0
ACTIVE SPATIAL RECEIVERS			404	7.8	1.85

**Figure 5.16.** Dynamic irradiance voxel calculation, on the left: rendered scene, on the right: active voxels during computation. The first line: all voxels are active, the second line: only voxels around dynamic objects are active.

In Figure 5.17 is an SSIM comparison of these two rendered images. As you see, images are equal, thus only unnecessary spatial receivers were omitted from the calculation.

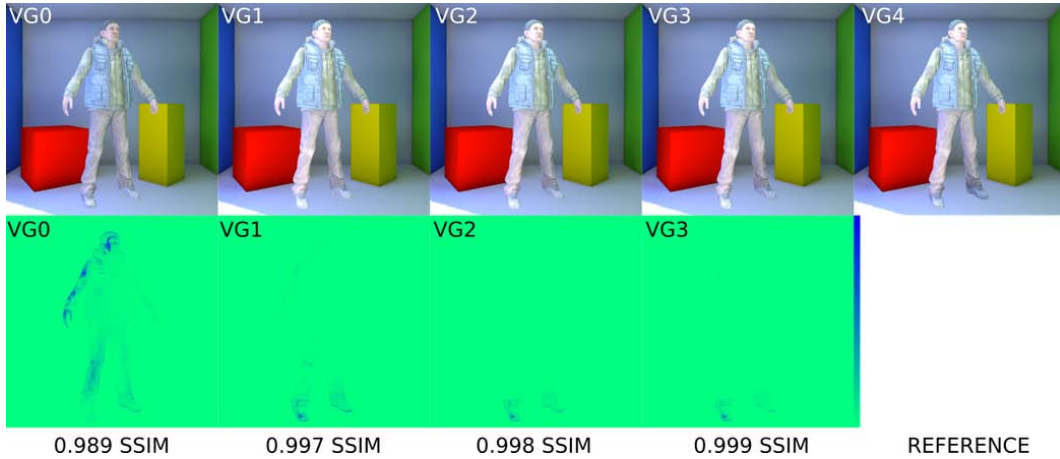


**Figure 5.17.** Dynamic irradiance voxel calculation, an SSIM difference of 1.0, complete match.

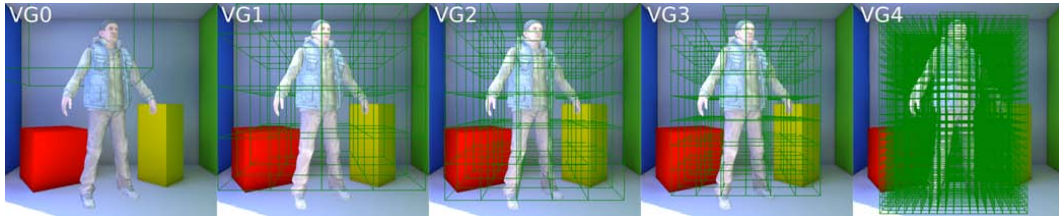
<sup>1</sup> [https://youtu.be/5A6B-\\_wSq14](https://youtu.be/5A6B-_wSq14)

### 5.2.1.2 Testing Voxel Grid - a Density of the Grid

Quality of shading is dependent on a density of the grid. To show how density can affect shading, I prepared several renders with a different grid density. Results are visible in Figure 5.18 in the first row. The second row shows how similar are other images to image 4 (highest density) in terms of the SSIM comparison. In Figure 5.19 are visible currently active voxels used for computation. If you compare the first image with the last one, there are clear differences. However, these examples use extreme setups. Density showed in pictures 1,2,3 is sufficient for generally good results.



**Figure 5.18.** Irradiance voxel grid with different density setups. The lowest density on the left, to the highest density on the right. Rendered images are in the first row, SSIM comparisons with image 4 are in the second row.



**Figure 5.19.** Irradiance voxel grid with different density setups, active voxels visualization.

Information about different setups is shown in Table 5.3.

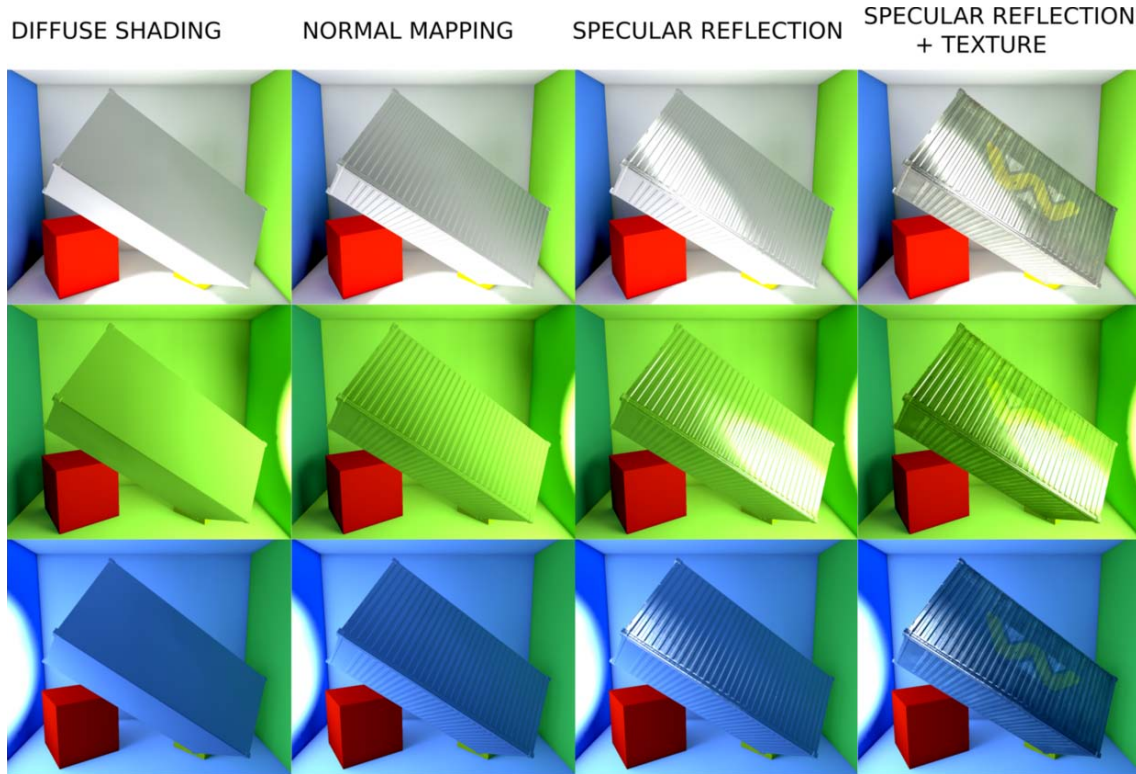
image	VG0	VG1	VG2	VG3	VG4
Spatial Receivers	8	180	800	1 440	59 976
active Spc. recs.	8	168	562	871	13 396
render time [ms]	4.15	4.19	4.33	4.44	15.70

**Table 5.3.** Table of density setup for irradiance voxel grid testing.



### 5.2.1.3 Layers of Shading

Let me firstly show separate layers of shading, see Figure 5.20. There are renders of a container in a Cornell box. The container model is dynamic. From the left to the right there is: only diffuse shading, diffuse shading with normal maps, diffuse + specular shading, all combined with a diffuse texture. A light source is a dynamic projective light with white emission color.



**Figure 5.20.** Layers of shading, from the left to the right: only diffuse shading, diffuse shading with normal maps, diffuse + specular shading, all combined with a diffuse texture.

#### 5.2.1.4 Shadow Casting from Dynamic Objects

I implemented dynamic lights using shadow mapping. Because we have visibility information even in run-time (if we render dynamic objects to the light source depth map) we can project this shadow on receivers with dynamic objects taken into account. This results in shadows cast from dynamic objects. Shadows are not ground truth, because precomputed visibility of receivers does not take any dynamic object into account, but it is a good and visually pleasing approximation.

Example of the dynamic object casting shadow is in Figure 5.21, where the dynamic object is a character in the middle.



**Figure 5.21.** Shadow casted by a dynamic object (character).

#### 5.2.1.5 Static Light Sources and Dynamic Objects

My approach supports not only dynamic light sources for dynamic objects, but also static precomputed light sources. There is no visibility information about the dynamic object for precomputed light sources, so the shadow reconstruction is not possible, but shading is possible nevertheless.

In Figure 5.22 you can see how static light sources illuminate a dynamic object (container). Notice how in the last image colors blend on the dynamic object surface.

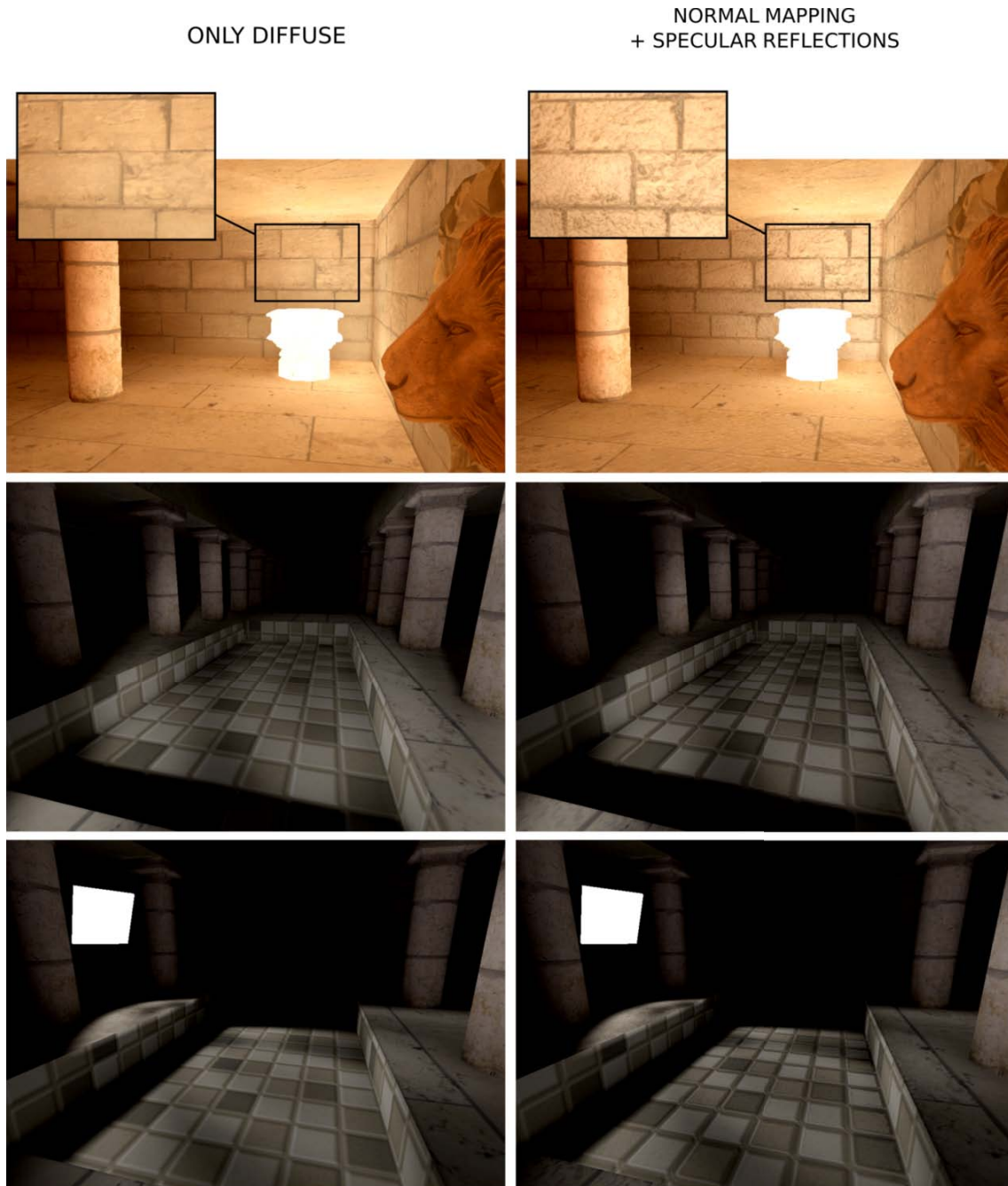


**Figure 5.22.** Static light sources illuminate a dynamic object.

### 5.2.1.6 Normal Mapping for Static Scene

As it was already mentioned in section 4.1, my extension supports approximation of normal mapping for a static scene without the need of radiance transport, just using the irradiance voxel grid.

Examples are shown in Figure 5.23.



**Figure 5.23.** Normal mapping for a static scene, the left image is without any normal mapping, the right image is with normal mapping.



## 5.2.2 Dynamic Level of Detail

I measured 3 different scenes, each for a different purpose. Testing hardware for this section was the following PC setup: 2× Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, RAM: 64GB, GPU NVIDIA GeForce GTX Titan Black 6GB, 980 Mhz.

After these measurements comes testing of each changeable LOD aspect (distance, SH degree, number of relight rays, camera view dependent zones, and pixel border extensions).

### 5.2.2.1 Open Hall

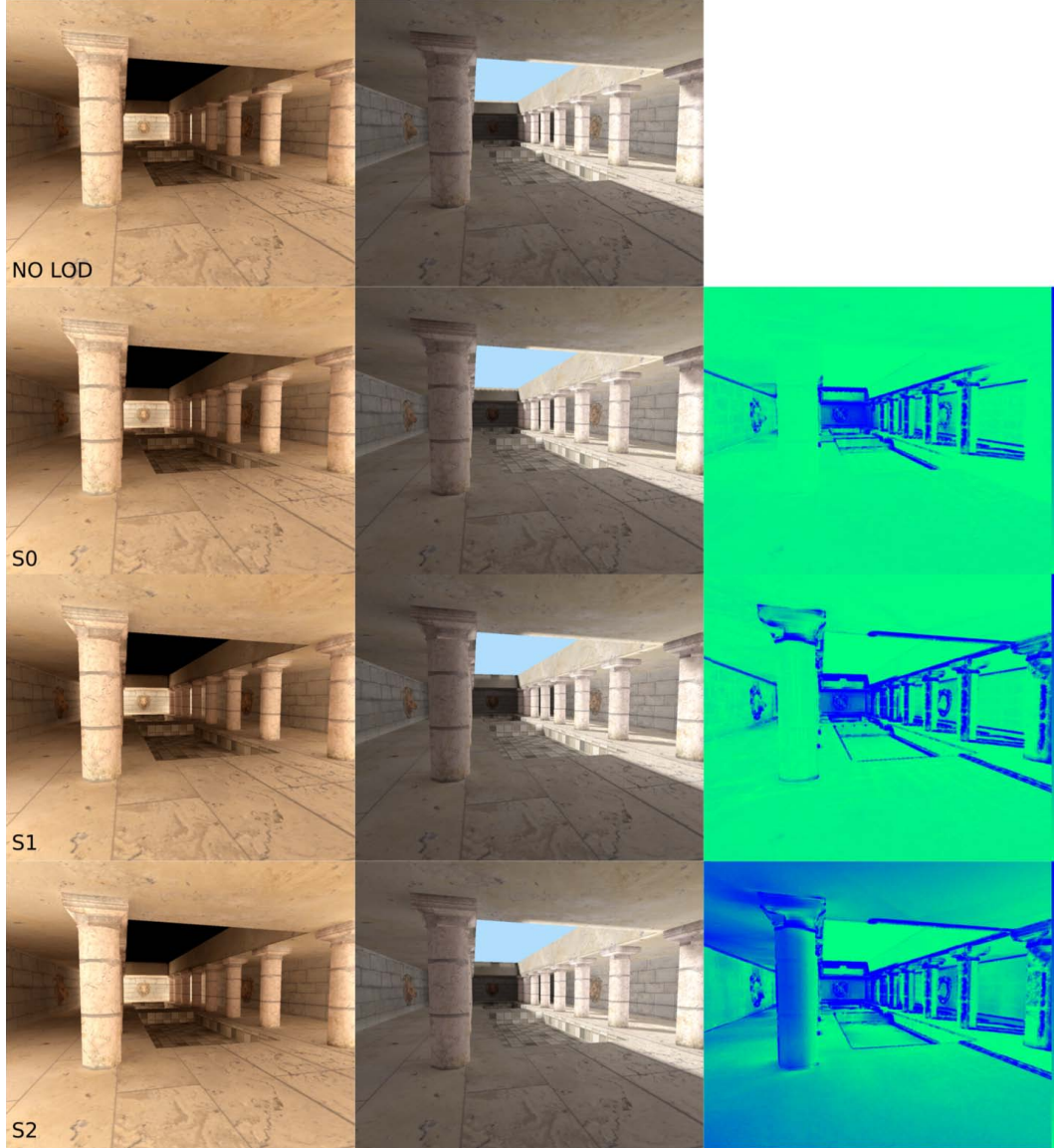
For this scene, I chose 4 different LOD setups, which are all visible in Table 5.4. As LOD also affects camera position and direction (because of frustum based leveling of detail), I chose 3 different camera setups. This scene was precomputed with a total number of 512 026 receivers and 44 probes.

setup	LVL 0	LVL 1	LVL 2	LVL 3
NO LOD	[7 , 0.0]	[7 , $\infty$ ]	[7 , $\infty$ ]	[7 , $\infty$ ]
S0	[7 , 0.0]	[6 , 20.0]	[4 , 50.0]	[3 , 100]
S1	[7 , 0.0]	[7 , 11.0]	[6 , 30.0]	[5 , 60]
S2	[4 , 0.0]	[4 , 11.0]	[4 , 30.0]	[3 , 60]

**Table 5.4.** Open Hall testing LOD setups: [SH degree, LOD distance].

Measured scenes with these setups are shown in figures 5.24, 5.25 and 5.26. The first line shows renders using a dynamic directional light and the second line the same renders but using a static area light.

Figure 5.24 presents renders for the first camera. This figure is followed by Table 5.5, where are measured times for each render. Measured times are the same for each row of Figure 5.24, because different light setup does not affect computation time. The third column in Figure 5.24 shows SSIM differences between NO LOD setup and others.

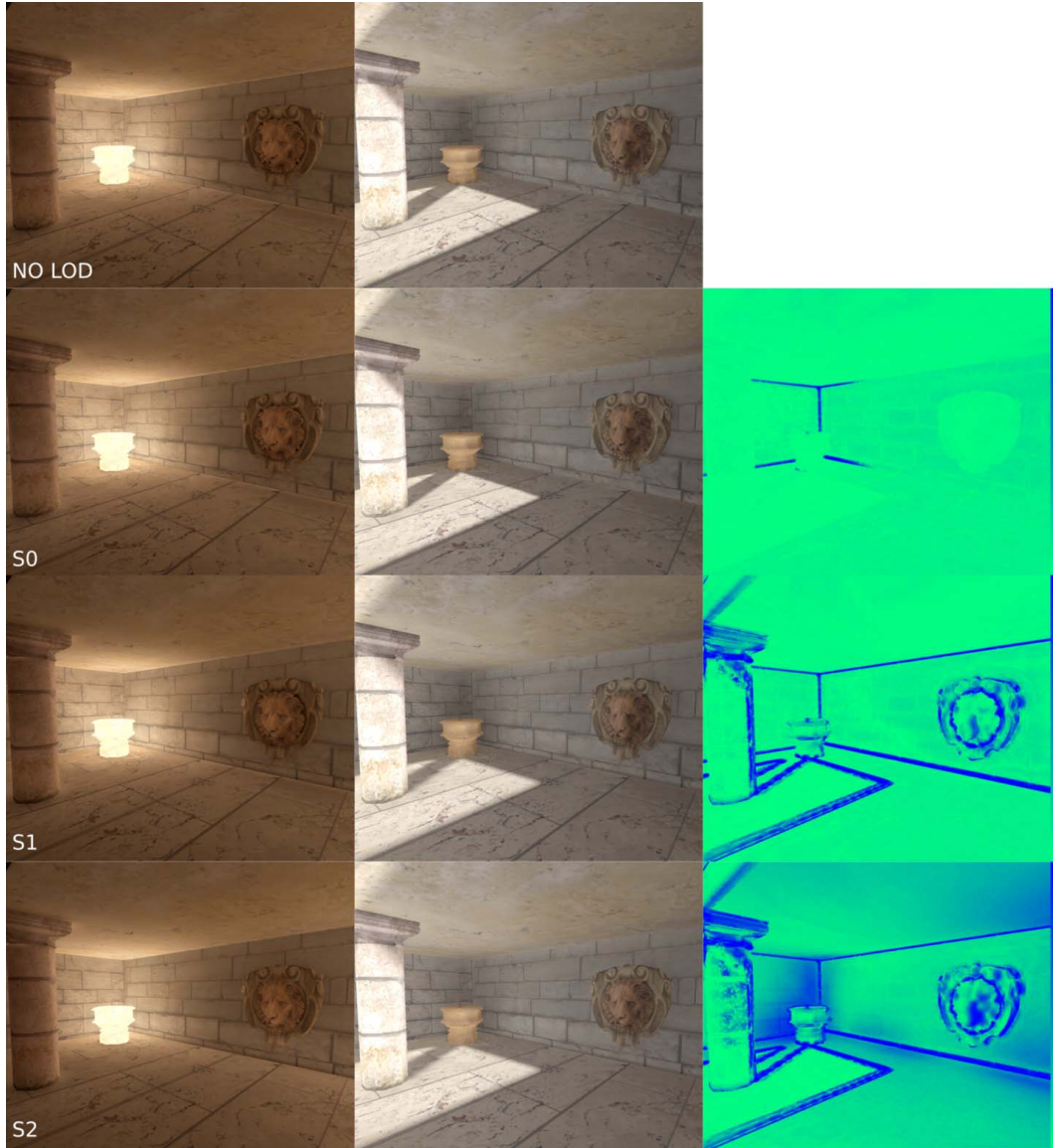


**Figure 5.24.** Rendered scene Open Hall, camera 1, in rows are from the top to the bottom setups from Table 5.4, the first line is without LOD. The third column is the SSIM difference between the image in the first line and image in the current line.

	active receivers	comp. time [ms]	speed-up	SSIM
NO LOD	512 026	1.99	1.0	1.0
S0	275 190	1.50	1.33	0.995
S1	158 371	1.35	1.47	0.991
S2	158 371	1.23	1.62	0.984

**Table 5.5.** Computation times for renders in Figure 5.24 with setups from Table 5.4.

Figure 5.25 contains renders for the second camera, the third column contains SSIM differences. The figure is followed by Table 5.6, where are measured times for each render.

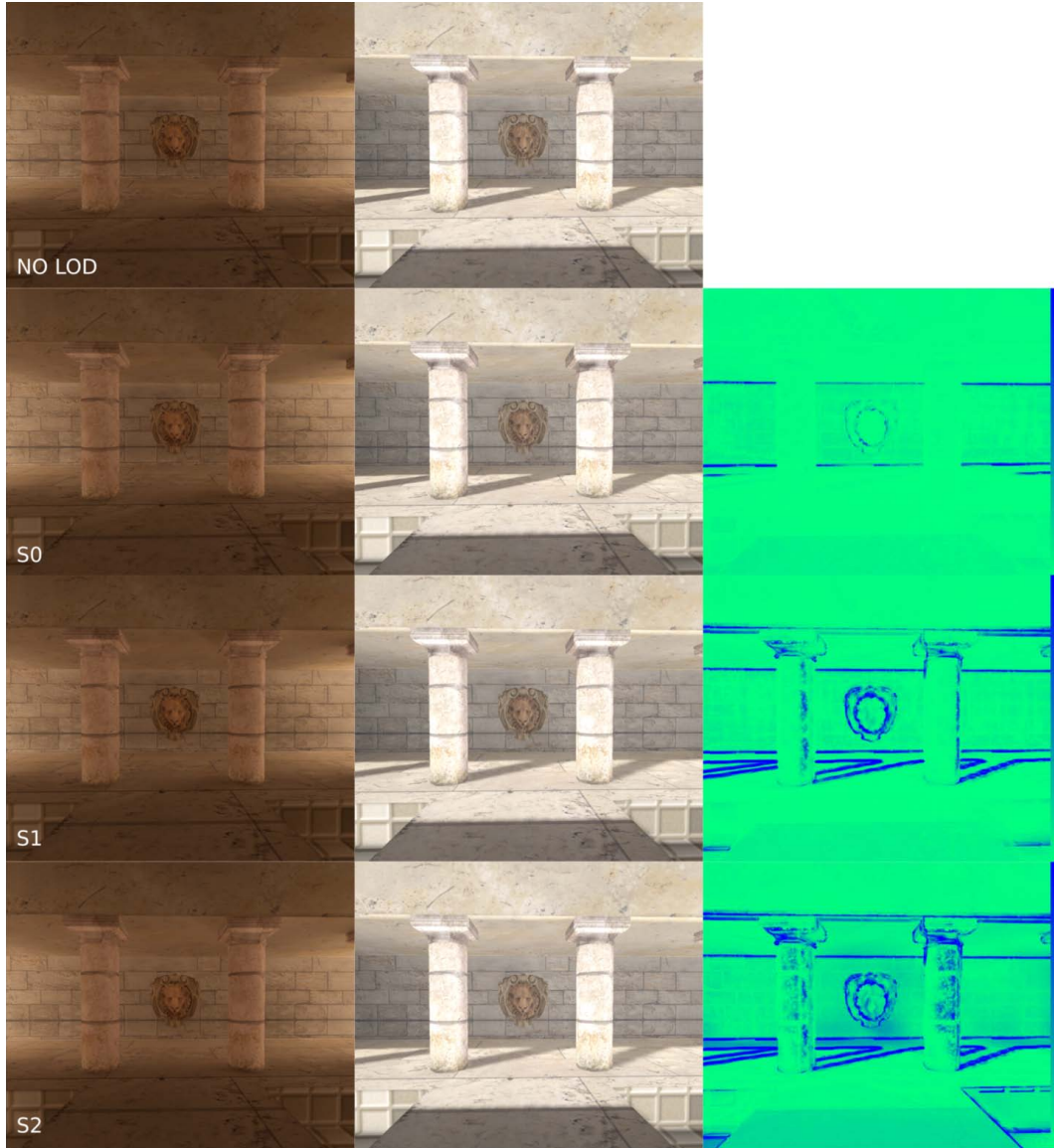


**Figure 5.25.** Rendered scene Open Hall, camera 2, in rows are from the top to the bottom setups from Table 5.4, the first line is without LOD. The third column is the SSIM difference between the image in the first line and image in the current line.

	active receivers	comp. time [ms]	speed-up	SSIM
NO LOD	512 026	1.99	1.0	1.0
S0	135 452	1.25	1.60	0.998
S1	99 744	1.24	1.60	0.993
S2	99 744	1.15	1.73	0.989

**Table 5.6.** Computation times for renders in Figure 5.25 with setups from Table 5.4.

Last Figure 5.26 contains renders for the third camera, the third column contains SSIM differences. The figure is followed by Table 5.7, where are measured times for each render.



**Figure 5.26.** Rendered scene Open Hall, camera 3, in rows are from the top to the bottom setups from Table 5.4, the first line is without LOD. The third column is the SSIM difference between the image in the first line and image in the current line.

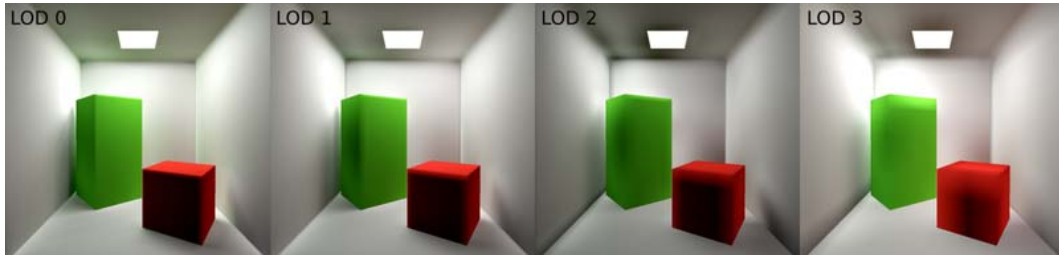
	active receivers	comp. time [ms]	speed-up	SSIM
NO LOD	512 026	1.99	1.0	1.0
S0	217 911	1.43	1.39	0.998
S1	126 279	1.27	1.57	0.996
S2	126 279	1.16	1.72	0.994

**Table 5.7.** Computation times for renders in Figure 5.26 with setups from Table 5.4.



### 5.2.2.2 Cornell Box

I tested this small Cornell Box for different LOD lightmap textures. Maximal lightmap size was  $512 \times 512$ . The first image is with a maximal lightmap size (LOD 0), the second image with a lightmap half the size (LOD 1) and so on. There are 4 LODs in total, so 4 images for 4 different lightmap sizes. The scene Cornell Box was precomputed with a total of 67 682 receivers and 7 probes. Images are shown in Figure 5.27.



**Figure 5.27.** Rendered scene Cornell Box with different lightmap size for different LOD. From the left to the right is LOD 0 to LOD 3.

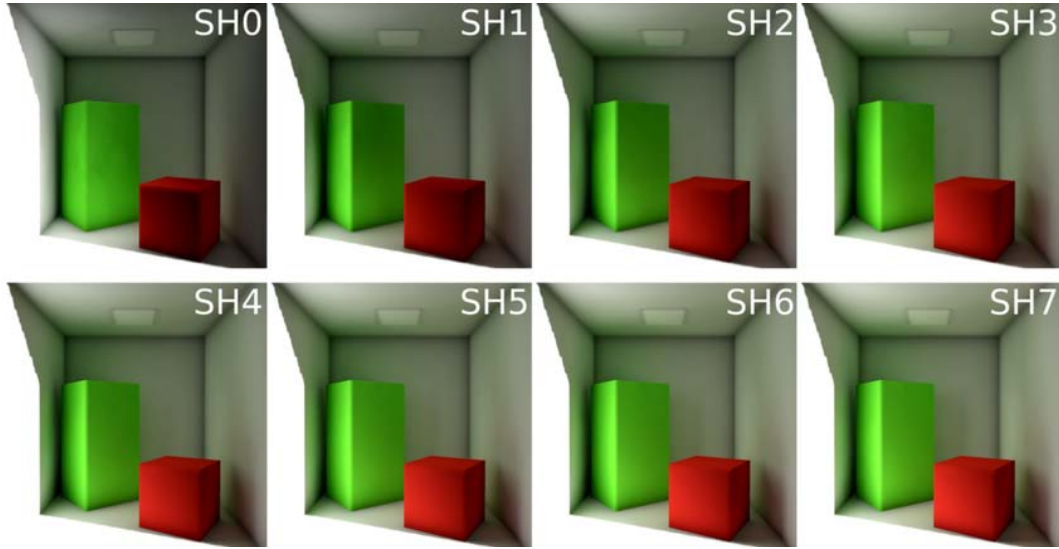
Computation times are shown in Table 5.8. As it can be expected, with half of the receivers, computation time is two times shorter.

	LOD 0	LOD 1	LOD 2	LOD 3
active receivers	67 682	16 857	4 210	1 073
computation time [ms]	0.92	0.44	0.24	0.17
speed-up	1.0	2.10	3.83	5.41

**Table 5.8.** Computation times for renders of Cornell Box scene.

### 5.2.2.3 Large Cornell Box

As this Cornell box provides noticeable color bleeding, I rendered this scene with different SH degree to compare how estimation of color bleeding using different degrees of SH will look like and how much time will it take. The scene Large Cornell Box was precomputed with 342 009 receivers and 13 probes. All renders with different degrees are shown in Figure 5.28.



**Figure 5.28.** Rendered scene Large Cornell Box with different SH degree for LOD 0 only.

In Table 5.9 are measured times for Large Cornell Box scene.

	SH 0	SH 1	SH 2	SH 3
computation time [ms]	1.42	1.42	1.43	1.43
speed-up	1.06	1.06	1.05	1.05

	SH 4	SH 5	SH 6	SH 7
computation time [ms]	1.43	1.43	1.46	1.50
speed-up	1.05	1.05	1.02	1.0

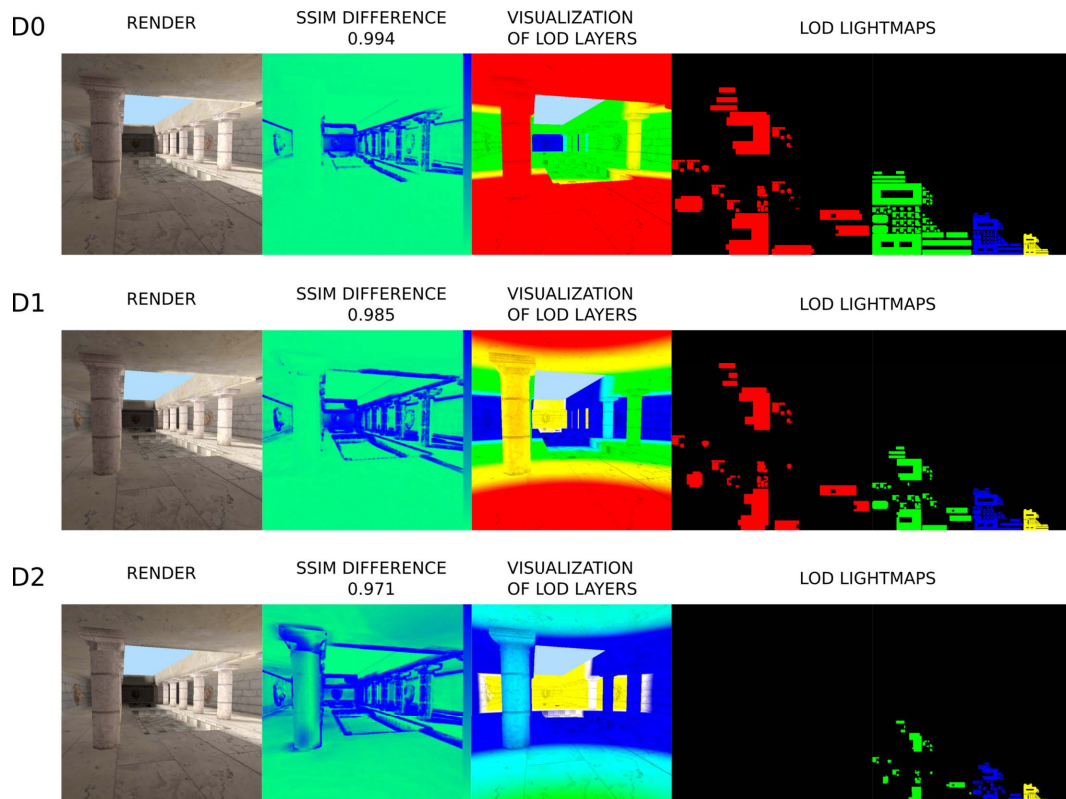
**Table 5.9.** Computation times, for renders of Large Cornell Box scene.

#### 5.2.2.4 Testing LOD Distance

The most significant value to change in my dynamic LOD system is definitely the drawing distance. The drawing distance change not only which the texture will appear on the screen, but also which sections of texture will be computed and which will be omitted. To demonstrate this, I prepared a few examples in Figure 5.29. Each line has the following distance setup for LODs. Setups are shown in Table 5.10.

Setup	LOD 0	LOD 1	LOD 2	LOD 3
D0	0.0	20.0	50.0	100.0
D1	0.0	14.0	24.0	46.0
D2	0.0	0.0	12.0	30.5

**Table 5.10.** Table of distances setup for LOD testing.



**Figure 5.29.** Different LOD distance setups. From the top to the bottom: dst\_0 , dst\_1, dst\_2. Setups are from Table 5.10. LOD 0, 1, 2, 3 is colored with red, green, blue, yellow color. In the first column is the rendered scene. In the second is SSIM comparison with the reference image. The third column shows visualized LODs with color. The fourth and the fifth columns are lightmaps for LODs 0, 1, 2, 3.

In Figure 5.29 you can see how different setup leads to uncalculated irradiance in lightmaps of different LODs. For example, in the first line, roughly half of the floor and ceiling is calculated in LOD 0. The rest is calculated in LOD 1 and 2. And in the second line no irradiance has been calculated in LOD 0 at all. This is because LOD 1 starts at 0.0. Thus, LOD 0 will never be calculated in this case.

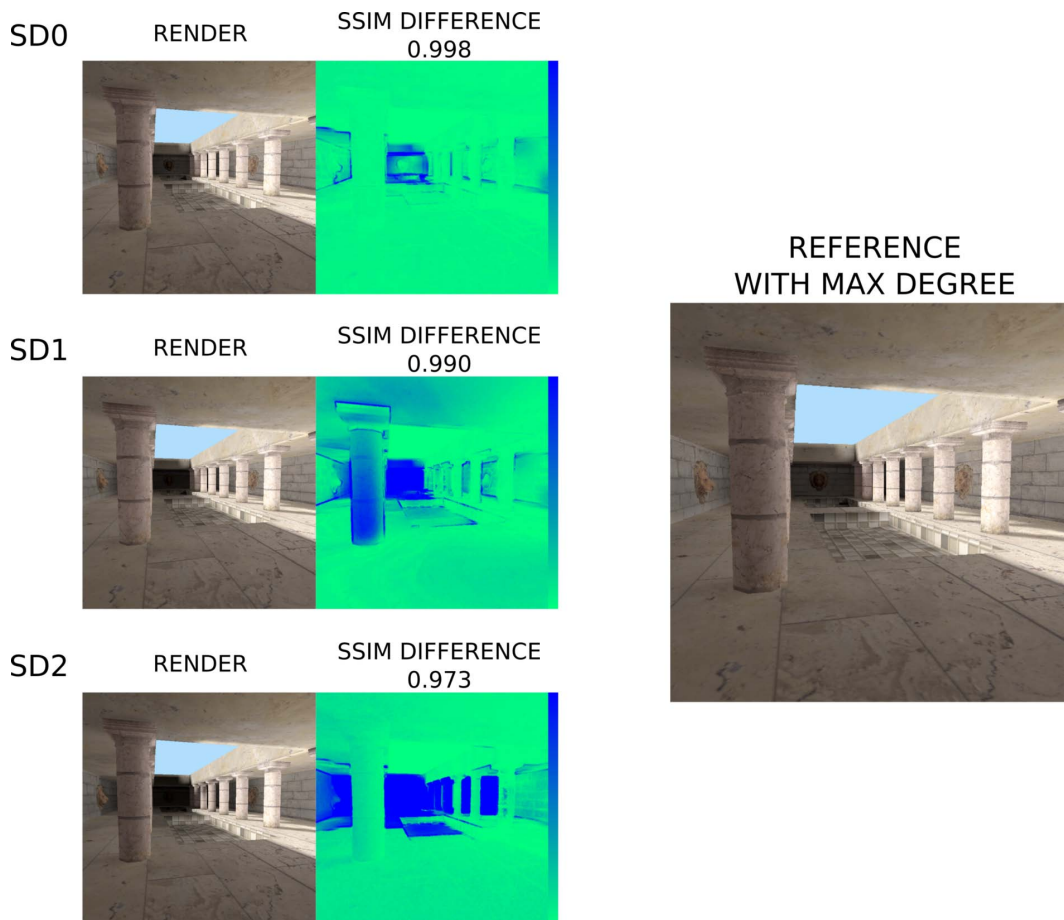
Furthermore, notice how lightmaps of different LODs are blend together at their borders. This is done to hide imperfections due to different resolutions of the textures. Images were compared with a reference image, which was rendered without LOD (every receiver and probe was in LOD 0).

### 5.2.2.5 Testing LOD Degree of Spherical Harmonics

Changing a degree of SH for different LODs got its advantages in a matter of computation speed-up. However, this change brings discontinuity to the system. If degrees of neighboring LODs are drastically different, it shows negatively on the resulting render. Few setups are shown in Figure 5.30. Setups themselves are noted in Table 5.11.

Setup	LOD 0	LOD 1	LOD 2	LOD 3
REFERENCE	7	7	7	7
SD0	7	5	3	2
SD1	4	3	2	0
SD2	7	2	0	0

**Table 5.11.** Table of SH degree setup for LOD testing.



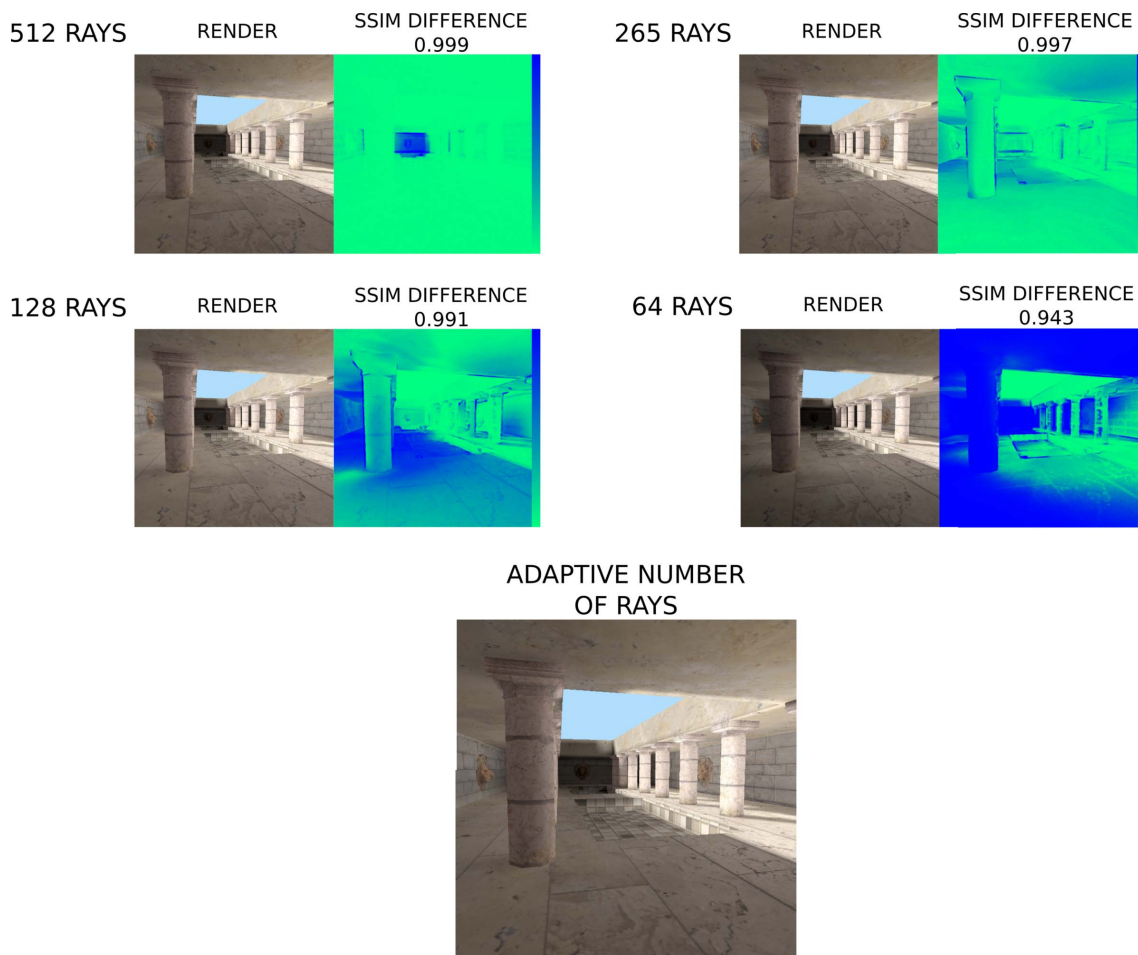
**Figure 5.30.** Different SH degree setups. Rendered results for setups from Table 5.11. A reference image is on the right. Rendered images are in the left column. SSIM differences, tested against the reference image (on the right), are in the right column.



In Figure 5.30 in the third row are clearly visible seams between LODs due to the big difference of the SH degree in neighboring LODs. During testing, I noticed that the maximal gap between LODs SH degree, which do not produce artifacts, is two. Greater gaps leave visible seams in the render.

### 5.2.2.6 Testing LOD Number of Relight Rays for Probes

For this testing I prepared a scene, where is clearly visible how dynamic LOD is able to blend together lightmaps which have been calculated with a different number of relight rays. Results are shown in Figure 5.31. Notice how a lower number of relight rays does not necessarily mean darker result. It just means that an approximation of a signal function will be less accurate. I measured the similarity of an image at the bottom (adaptive solution) with the rest of the images. As you can see, similarities show that the adaptive number of relight rays matches the most with the highest number of relight rays, which was desired. As the number of rays decreases with a bigger distance, similarities disappear.



**Figure 5.31.** A different number of relight rays setups. The bottom image shows render with an adaptive number of relight rays, for LOD 0, 1, 2, 3 algorithms used 512, 256, 128, 64 relight rays. The rest of the images has a static number of relight rays. SSIM differences were tested against image using the adaptive method (at the bottom).

### 5.2.2.7 Testing Camera View Dependant Zones in LOD

Previously shown in chapter 4.2 (Figure 4.8), LOD is capable of discarding calculations even in dependency to a camera view. An example is shown in Figure 5.32.



**Figure 5.32.** Camera View Dependant Zones in LOD. On the left: NO camera view dependant zones, 229 076 computed receivers. In the middle: WITH camera view dependant zones, 105 259 computed receivers. On the right is the SSIM comparison. A complete number of receivers is 335 015.

### 5.2.2.8 Testing Lightmap Border Extension Pixels in LOD

As it was mentioned in chapter 3.2.1, due to the linear filtering, edges of the lightmap blend together with an initial black color of the lightmap. This creates dark edges on meshes during rendering. Artifact gets even worse if LOD is used because smaller lightmap will get stretched out to the bigger area. Thus, blending is even further into a mesh. To solve this problem, I extended irradiance at pixels laying on borders. After LOD was implemented, I had to implement this feature even to the lower LODs of the lightmap. Problem and its solution are shown in Figure 5.33.



**Figure 5.33.** Pixel border extension, from the left to the right: No border extension, border extension only in LOD 0, border extension in all LODs. Wrong color blending is highlighted.

## Chapter 6

### Conclusion

In the first part of my work, I described my reimplementation of the method for real-time global illumination [4]. I split the implementation into two applications: the precomputation application and the real-time rendering application. The precomputation application takes mesh as input together with several user-defined constants and computes all textures and data needed for real-time rendering. I invented several algorithms to make this application fully automatic: automatic placement of meshes into the lightmap, automatic scaling of the lightmap (chapter 3.2.1) and automatic calculation of the initial set of probes (chapter 3.2.2). The real-time rendering application uses the output from the precomputation application and it renders the scene in real-time using OpenGL.

In the second part of my work, I created two extensions to the proposed algorithm [4]. The first extension is able to shade dynamic objects in real-time with global illumination from the scene using spatial irradiance receivers in a sparse voxel grid. Furthermore, my method is able to approximate normal mapping and specular reflections for both dynamic objects and static scene without the need for costly radiance transport. This method even supports shadow-casting by dynamic objects when dynamic lights are used.

The second extension uses a dynamic Level of Detail to omit computations and improve performance. My dynamic Level of Detail provides multiple lightmaps. One lightmap for every Level of Detail. Each lightmap has different resolution and each Level of Detail has different detail settings. Every aspect of my dynamic Level of Detail method can be changed in run-time.

I have tested all the important constants and implementation decisions of the proposed method in chapter 5.1. I have also tested my two extensions in chapter 5.2.

For the demonstration of the proposed method [4] and my extensions, I have implemented the precomputation application and the real-time rendering application.

### 6.1 Future Work

As the method is able to cast shadows for dynamic objects only from dynamic light-sources, I want to focus on the full support of shadow-casting even from indirect illumination. Another field of study would be high-frequency spatially-varying BRDF as it is not well handled by the current method.

Another improvement can be done in the precomputation part in terms of speed. As most of the computation can be done in parallel, the precomputation application would greatly benefit from multithreaded or GPU implementation.



## References

- [1] Lafortune, Eric P. and Willems, Yves D., Bi-directional path tracing, Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93), pages 145–153, December 1993, Alvor, Portugal
- [2] Stefan Brabec and Thomas Annen and Hans-Peter Seidel, Practical Shadow Mapping. Journal of Graphics Tools, vol.7,num.4,pages 9-18, 2002 Taylor & Francis
- [3] Ravi Ramamoorthi, Precomputation-Based Rendering, vol.3, num. 4, pages 281-369, Foundations and Trends® in Computer Graphics and Vision 2009
- [4] Ari Silvennoinen, Jaakko Lehtinen., Real-time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes. ACM Transactions on Graphics, Vol. 36, No. 6, Article 230. Publication date: November 2017.
- [5] Henry Schäfer and Jochen Süßmuth and Cornelia Denk and Marc Stamminger, Memory efficient light baking. vol. 36, num. 3, pages 193 - 200, Computers & Graphics 2012
- [6] Ritschel, Tobias and Dachsbacher, Carsten and Grosch, Thorsten and Kautz, Jan. The State of the Art in Interactive Global Illumination. vol.31, num.1, pages 160-188, Computer Graphics Forum 2012
- [7] GORAL C. M., TORRANCE K. E., GREENBERG D. P.,BATTAILE B. Modeling the interaction of light between diffuse surfaces. ACM SIGGRAPH Computer Graphics 18, 3 (1984), 213–222.
- [8] NICHOLS G., SHOPF J., WYMAN C.: Hierarchical Image-Space Radiosity for Interactive Global Illumination. Computer Graphics Forum 28, 4 (2009), 1141–1149. 5, 11, 15
- [9] JENSEN H. W.: Global illumination using photon maps. In Proc. EGWR (1996), pp. 21–30. 5, 15
- [10] KELLER A.: Instant radiosity. Proc. SIGGRAPH 31, 3 (1997), 49–56. 6, 25
- [11] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P. Lightcuts: a scalable approach to illumination. ACM Trans. Graph. (Proc. SIGGRAPH) 24, 3 (2005), 1098–1107. 8, 12, 15
- [12] CHRISTENSEN P.: Point-Based Approximate Color Bleeding. Tech. rep., Pixar, 2008. 6, 8, 15, 25
- [13] CHANDRASEKHAR S.: Radiative Transfer. Dover Pubns, 1950. 9
- [14] KAPLANYAN A., DACHSBACHER C.: Cascaded light propagation volumes for real-time indirect illumination. In Proc. I3D (2010), p. 99. 9, 10, 15, 26
- [15] DACHSBACHER C., STAMMINGER M.: Reflective shadow maps. In Proc. I3D (2005), p. 203. 6, 7, 9, 11, 12, 14, 15, 25
- [16] Peter-Pike Sloan, Jan Kautz, and John Snyder, Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Enviroments.

ACM Transactions on Graphics, Vol. 21, Issue 3, Pages 527 - 536. Publication date: July 2002.

- [17] R. T. Seeley: Spherical Harmonics. The American Mathematical Monthly, vol. 73, num. 4P2, o. 115-121, Taylor & Francis 1966
- [18] Robin Green, Spherical Harmonics Lighting: The Gritty Details. Sony Computer Entertainment America, January 16, 2003.
- [19] LIU X., SLOAN P.-P., SHUM H.-Y., SNYDER J.: All-Frequency Precomputed Radiance Transfer for Glossy Objects. In Proc. EGSR (2004), vol. 3, pp. 337-344. 10, 26
- [20] SLOAN P.-P., HALL J., HART J., SNYDER J.: Clustered principal components for precomputed radiance transfer. Proc. SIGGRAPH 22, 3 (2003), 382. 10, 11, 12
- [21] LEHTINEN J., KAUTZ J.: Matrix radiance transfer. In Proc. I3D (2003), I3D '03, pp. 59-64. 10
- [22] GREGER G., SHIRLEY P., HUBBARD P., GREENBERG D.: The irradiance volume. IEEE Computer Graphics and Applications 18, 2 (1998), 32-43. 10, 26
- [23] KONTKANEN J., LAINE S.: Ambient occlusion fields. In Proc. I3D (2005), p. 41. 10, 11, 26
- [24] PHARR, Matt; GREEN, Simon. Ambient occlusion. GPU Gems, 2004, 1: 279-292.
- [25] PAN M.: Precomputed Radiance Transfer Field for Rendering Interreflections in Dynamic Scenes. Computer Graphics Forum 26, 3 (2007), 485-493. 10
- [26] Johannes Jendersie, David Kuri, and Thorsten Grosch. 2016. Precomputed illuminance composition for real-time global illumination. In Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '16). ACM, New York, NY, USA, 129-137. DOI: <https://doi.org/10.1145/2856400.2856407>
- [27] Peter-Pike Sloan, Efficient Spherical Harmonic Evaluation. Journal of Computer Graphics Techniques, 2 (2), 2013, <https://www.ppsloan.org/publications/SHJCGT.pdf>
- [28] Adekitan, A.I.. (2014). MONTE CARLO SIMULATION.
- [29] Amanatides and Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. Eurographics 1987.



## Appendix A

### The Application Guide

All controls of the provided application are explained in this appendix. Prepared examples can be executed by running provided scripts (.sh for Linux).

#### A.1 Scene Configuration File

The application binary takes as an input a configuration file, file with `.scene` suffix. From this file, all parameters can be set up. All commands start with `!` followed by command name and arguments, comments can be added by `#` at the beginning of the comment. List of all commands is provided on the next page in Figure A.1.

command	description
<b>!SSF *.ssf</b>	Load precomputed static scene from the provided file. REQUIRED.
<b>!DOBJ *.obj</b>	Load a dynamic object.
<b>!DOBJ_GRID_INSERT 1</b>	Enlight the dynamic object with the irradiance grid?
<b>!DOBJ_SCALE 1</b>	Set scale of the dynamic object.
<b>!DOBJ_INIT_TRANSLATE 10.5 0.2 4</b>	Translate the dynamic object to an initial position.
<b>!DOBJ_INIT_ROTATE 20.0 10 5 1</b>	Rotate the dynamic object to an initial position (angle , vec3(axis)).
<b>!DOBJ_ANIM_TRANSLATE 1 0 2</b>	Translate the dynamic object in every frame by this vector.
<b>!DOBJ_ANIM_ROTATE 0.01 1 0.2 1</b>	Rotate the dynamic object in every frame by this angle around this axis (angle , vec3(axis)).
<b>!DOBJ_BEZIER -1 3 0; -0.9 8.5 -6.1;</b>	Move the dynamic object along the Bezier curve defined by the argument points.
<b>!DOBJ_BEZIER_SPEED 0.002</b>	Set a speed of the dynamic object movement along the Bezier curve.
<b>!BACKGROUND_COLOR 0 0 0</b>	Set color of the background (R G B).
<b>!SCREEN_RESOLUTION 1024 720</b>	Set an initial screen resolution (width height).
<b>!MAT_EMIS 3 0.2</b>	Set an emissivity of material 3 to 0.2.
<b>!LIGHT_MODE AREA</b>	Set a mode of the light: AREA, FLASH, POINT.
<b>!LIGHT_INIT_TRANSLATE 12 3 11</b>	Translate the light to an initial position.
<b>!LIGHT_INIT_YAW 12.1</b>	Set an initial yaw for the light.
<b>!LIGHT_INIT_PITCH -4</b>	Set an initial pitch for the light.
<b>!LIGHT_INIT_COLOR 0 0 0</b>	Set an initial color for the light (R G B).
<b>!CAMERA_INIT_TRANSLATE 5 0 1</b>	Translate the camera to an initial position.
<b>!CAMERA_INIT_YAW 180</b>	Set an initial yaw for the camera.
<b>!CAMERA_INIT_PITCH -0.5</b>	Set an initial pitch for the camera.
<b>!CAMERA_FOV 60</b>	Set a field of view for the camera.
<b>!TONE_MAPPING E</b>	Set an initial tone mapping to: E (exposure), F (filmic), R (Reinhard).
<b>!TONE_MAPPING_GAMMA 1.58</b>	Set an initial tone mapping gamma.
<b>!TONE_MAPPING_EXPOSURE 11.9</b>	Set an initial tone mapping exposure (only for exposure tone mapping).
<b>!TONE_MAPPING_CONTRAST 2.10</b>	Set an initial tone mapping contrast.
<b>!TONE_MAPPING_DIMMER 0.03</b>	Set an initial tone mapping dimmer.
<b>!SCREENSHOT_FOLDER ./screen_shots</b>	Set a folder for the screenshots.
<b>!LOD_DISTANCE 0.0 20.0 50.0 100.0</b>	Set LOD distances: lod 0, lod 1, lod 2, lod 3 (lod 0 always 0.0).
<b>!LOD_SH_DEGREE 7 4 3 1</b>	Set SH degree for LOD: lod 0, lod 1, lod 2, lod 3.
<b>#Commented line</b>	Comment a line using "#".

**Figure A.1.** List of all scene configuration file commands.



## A.2 Application GUI

The application provides GUI to change several parameters in run-time. Press **F** to enable camera movement with mouse, press **L** to switch between camera and light source movement. Use **W, A, S, D** to move around with the camera or the light source. Press **T** to switch between 3D view and texture view. Press **1** to display LOD 0 lightmap, press **2** to display the rest of the lightmaps, press **3** to display shadow map. Change camera speed by **Page Up** and **Page Down** keys.

GUI is split into the left and the right panel. Press **G** to show/hide the left panel, press **H** to show/hide the right panel. Listing all features in the left panel:

Reset Camera - resets the camera to its initial position

Reflection - sets static reflection value for global illumination

Static Materials - emissivity for any material of a static mesh can be set up here

Dynamic Light - set mode, color and rendering of impostor object for the dynamic light

Tone Mapping - setup of tone mapping

Visualization - check certain box to visualize probes, bounding volumes or voxels

Information - press to display information about the static scene

Listing all features in the right panel:

Level of Detail - set every layer of LOD separately: Level 0,1,2,3

Distance - set drawing distance of certain LOD lightmap

SH Degree - set a degree of Spherical Harmonics for certain LOD

Reset Settings - reload initial LOD setup

Measurements - display window for measuring time and number of active receivers

## Appendix B

### Used Libraries and Extern Packages

Intel Embree<sup>1</sup>  
Efficient Spherical Harmonic Evaluation<sup>2</sup>  
SVD C++ implementation <sup>3</sup>  
OpenGL <sup>4</sup>  
Nuklear GUI<sup>5</sup>  
OpenGL Mathematics<sup>6</sup>  
GLFW<sup>7</sup>  
The OpenGL Utility Toolkit<sup>8</sup>  
The OpenGL Extension Wrangler Library<sup>9</sup>  
Assimp <sup>10</sup>  
DevIL <sup>11</sup>  
NVIDIA CUDA<sup>12</sup>  
NVIDIA OptiX<sup>13</sup>

---

<sup>1</sup> <https://www.embree.org/>  
<sup>2</sup> <http://jcgt.org/published/0002/02/06/>  
<sup>3</sup> <http://svn.lirec.eu/libs/magicsquares/src/SVD.cpp>  
<sup>4</sup> <https://www.opengl.org/>  
<sup>5</sup> <https://github.com/vurtun/nuklear>  
<sup>6</sup> <https://glm.g-truc.net/0.9.9/index.html>  
<sup>7</sup> <https://www.glfw.org/>  
<sup>8</sup> <http://freeglut.sourceforge.net/>  
<sup>9</sup> <http://glew.sourceforge.net/>  
<sup>10</sup> <http://www.assimp.org/>  
<sup>11</sup> <http://openil.sourceforge.net/>  
<sup>12</sup> <https://developer.nvidia.com/cuda-zone>  
<sup>13</sup> <https://developer.nvidia.com/optix>

## Appendix C

### Compilation of Provided Applications

Provided source code of the applications can be built on Linux systems. If one desires to compile the code on Windows, focus on `getline()` function as it has different requirements on Windows and type `ssize_t` which is not available on Windows.

There are three different applications provided. All of them can be compiled using `cmake`. To compile `PRECOMPUTATION_APP` and `SH_COMP_APP` use `gcc 7.3.0` or later. Use the following commands to compile one of these applications:

```
cd < source_code_folder >
```

```
cmake CMakeLists.txt
```

```
make
```

To compile `REAL_TIME_APP` use `gcc` of version less than 6.0 (I used `gcc 4.8`). As the `REAL_TIME_APP` uses `OptiX` together with `NVIDIA CUDA`, `nvcc` compiler is required. The `gcc` version requirement must hold due to `CUDA` compiler compatibility. Use the following commands to compile this application:

```
cd < source_code_folder >
```

```
cmake CMakeLists.txt
```

```
make
```

## Appendix D

### The Contents of The Enclosed DVD

