University of Magdeburg
School of Computer Science

Master’s Thesis

Spherical Illuminance Composition for Real-Time Indirect Illumination

Author:
David Kuri

April 29, 2015

Advisors:
Jun.-Prof. Dr. Thorsten Grosch
Department of Simulation and Graphics

Prof. Dr. Holger Theisel
Department of Simulation and Graphics
Kuri, David:
Spherical Illuminance Composition for Real-Time Indirect Illumination
Abstract

In photorealistic rendering, the simulation of global illumination is of great perceptual importance for the generation of convincing imagery. The concepts of light transport for the purpose of rendering are well understood, but expensive to calculate. For real-time solutions, simplification is necessary, often at the cost of visual quality.

We present a new real-time algorithm for the calculation of global illumination in diffuse scenes. A precomputation step allows for high visual quality with an infinite number of light bounces. Dynamic objects can receive indirect light and don’t show temporal artifacts. The proposed technique supports full dynamic lighting and works with all commonly used light source models. In addition, area and environment lighting are facilitated.

Furthermore, we present details on how our technique can be implemented on contemporary hardware. Various approaches are explained and compared to give guidelines for practical implementation.
Acknowledgements

I would like to express my thanks to my supervisors Jun.-Prof. Dr. Thorsten Grosch and Prof. Dr. Holger Theisel for reviewing this thesis. In addition, M. Sc. Johannes Jendersie has been a great help all throughout the involved research and was very enthusiastic about my work.

I would further like to thank Philipp Krause and the great people at Piranha Bytes where I developed a large part of the presented technique. In the short time I spent in their office, I learned a lot from the incredibly proficient developers and it was delightful to see how so many smart people spend their time and talent to create an utterly complex software system, just for people - myself included - to enjoy.

Lastly, I would like to thank my family, who have supported me greatly during my work on this thesis and the related changes in my life.
Contents

1 Introduction ............................................. 1
  1.1 Motivation ........................................... 1
  1.2 Scope of this thesis ................................... 2
  1.3 Outline ............................................. 3

2 Global illumination foundations ......................... 5
  2.1 Direct lighting ....................................... 5
  2.2 Photometric quantities ............................... 6
    2.2.1 Radiant and luminous energy .................... 6
    2.2.2 Luminous flux .................................... 6
    2.2.3 Luminous exitance ............................... 7
    2.2.4 Luminous intensity .............................. 7
    2.2.5 Luminance ....................................... 7
    2.2.6 Illuminance ..................................... 8
  2.3 Rendering equation ................................... 8
  2.4 BRDF ............................................. 8
  2.5 Path tracing ....................................... 9

3 Related work ........................................... 11
  3.1 Spherical harmonics .................................. 11
  3.2 Precomputed radiance transfer ...................... 14
    3.2.1 Overview ....................................... 14
  3.3 Radiosity .......................................... 15
    3.3.1 Overview ....................................... 15
    3.3.2 Real-time radiosity ............................. 16
  3.4 Light propagation volumes ........................... 16
  3.5 Voxel cone tracing .................................. 17

4 Algorithm ............................................. 19
  4.1 Overview .......................................... 19
  4.2 Precomputation ..................................... 20
    4.2.1 Surfel placement ................................ 20
    4.2.2 Surface group clustering ....................... 21
    4.2.3 Light probe placement .......................... 22
    4.2.4 Light probe calculation ....................... 23
  4.3 Runtime calculations ................................ 25
    4.3.1 Surfel and surface group update ............... 25
    4.3.2 Light probe and data structure update ........ 26
4.3.3 Evaluating the SH ........................................ 27

5 Implementation ............................................. 29
5.1 Precomputation implementation .......................... 29
  5.1.1 Surfel placement .................................... 29
  5.1.2 Surface group clustering ............................ 30
  5.1.3 Light probe placement .............................. 31
  5.1.4 Light probe calculation ............................ 31
5.2 Runtime implementation ................................ 32
  5.2.1 Pure CPU solution ................................. 33
  5.2.2 Hybrid solution .................................... 34
  5.2.3 Pure GPU solution .................................. 36

6 Evaluation .................................................. 39
6.1 Results .................................................. 39
6.2 Performance ............................................ 40
  6.2.1 Computation time .................................. 43
  6.2.2 Memory consumption .............................. 44
6.3 Limitations .............................................. 45

7 Conclusion ............................................... 47
  7.1 Future Work .......................................... 47

Appendix A .................................................. 49

Bibliography ................................................ 51
1. Introduction

This chapter describes the purpose, goal and structure of the thesis. Fundamental terms of computer graphics are introduced to provide context, and the contribution is outlined to simplify the understanding of the following chapters.

1.1 Motivation

Computer generated imagery of three-dimensional environments or objects is in high demand in a variety of industries. Especially the generation of images that are perceived to be realistic by a human observer is just as difficult as it is desirable. This is the focus of an area of computer graphics called photorealistic rendering.

In many cases, images need to be rendered on demand to reflect user input or other unpredictable events. These images or frames are shown in rapid succession to give the impression of motion, imposing a constraint on the time the generation of a single image is allowed to take. Real-time rendering is the generation of images fast enough to allow for immediate feedback to user input.

For example, in computer games the player can control certain elements of the game which affect the visual output. For this interaction to feel natural, an average of 30 or 60 frames per second (FPS), i.e. a maximum of approximately 33.3 or 16.6 ms per frame respectively, is a commonly posed requirement.

Other industries such as the movie industry are also affected by this. In contrast to games, the images of an animated movie are rendered offline, i.e. in advance with no hard time constraints. Nevertheless, artists benefit greatly from fast and responsive preview mechanisms during production.

The correct simulation of light and surfaces is of vital importance for photorealistic rendering. This seems like a simple problem at first. Light is emitted from light sources, partially absorbed, refracted and reflected by surfaces, until it eventually hits the observer. Neither modeling light sources nor surfaces is a trivial task. However, computing vast numbers of light paths in reasonable time, let alone under real-time constraints, is exceptionally hard.
1. Introduction

Figure 1.1: Comparison between local illumination (with ambient term and perfect reflection on mirroring objects) and global illumination. Ambient occlusion is most noticeable in the corners. Caustics from the glass sphere can be observed on the left wall and floor.

The realistic simulation of light, including indirect light that was at least once reflected on any surface of the 3D scene, is called global illumination (GI). Various observable effects are produced:

- **Ambient light.** Light will be reflected until it is fully absorbed. This reflected light will eventually reach most surfaces of the scene. Pitch-black areas are extremely rare.

- **Ambient occlusion.** Reflected light is less likely to reach heavily occluded surfaces such as corners, small gaps or creases. They will be notably darker, providing structure and depth to the image.

- **Color bleeding.** Indirect light is colored by the surface it is reflected on. This can result in noticeable color bleeding from one surface onto another.

- **Caustics.** Due to reflection or refraction by curved objects, many light rays are focused and produce strong highlights.

To alleviate the problem of lighting computation, computer games have traditionally used local illumination. Using simple models for light sources and surface materials, the direct illumination of a surface point can be calculated easily. Indirect illumination is ignored. To avoid unrealistic pitch-black areas, a constant ambient light term is often added which increases the overall brightness of the scene but cannot account for the other effects of GI described above.

1.2 Scope of this thesis

The purpose of this thesis is twofold. First, existing GI techniques are reviewed and a new method for the calculation of indirect lighting in dynamic 3D scenes is developed. Second, various implementation structures are suggested and compared to give guidelines on how to use the algorithm in practice.
The proposed method is fast enough to be suitable for real-time applications on contemporary consumer hardware. It is based on the radiosity technique [GCT86] with various adaptions to speed up the computation. An infinite number of light bounces can be calculated iteratively. Indirect light is stored in the form of spherical harmonics (SH). This directional representation increases the quality of the results and enables the use of normal mapping. The idea of irradiance volumes [GSHG98] has been incorporated to provide indirect lighting for dynamic objects.

As stated above, the problem of GI is currently not feasible to solve under the given time constraints. Certain restrictions and approximations have to be accepted to arrive at a real-time solution. While staying true to the physical foundations of GI, a deviation from the correct solution is accepted as long as the visual results are persuasive.

For indirect lighting calculations, all involved surfaces are assumed to be purely diffuse as defined by Lambert. The luminous intensity emitted by the surface is independent of the observer’s angle of view, which greatly simplifies calculations.

Even in dynamic environments such as computer games a large percentage of the scene data does not change at runtime. Various methods make use of this by relocating parts of the algorithm into a precomputation step. The presented solution uses static geometry as both sender and receiver of indirect light while dynamic objects are treated as receivers only. This allows for partial precomputation which is necessary to achieve high quality results while staying within the given time constraints.

Indirect lighting information is typically ‘blurry’, i.e. of low frequency. To exploit that, indirect lighting is stored separately. Direct lighting is computed using traditional techniques of computer graphics. Indirect lighting is derived from a sampling of direct lighting information. That means that any kind of light source that can be used for direct lighting is easily used with the proposed technique to contribute to GI. Furthermore, lighting by inherently complex area lights and environment or sky lights is facilitated.

The algorithm scales well for large scenes. Using the proposed implementation structures, computation time increases gracefully with the total surface area and the extents of the scene. The proposed solution is very flexible and can be decoupled from the rendering loop. This can be used to e.g. calculate distant results less frequently and facilitate the handling of larger scenes. Furthermore, per-pixel calculations are cheap so that large screen resolutions are supported.

1.3 Outline

This thesis is structured into three parts. First, the required background knowledge is depicted. The foundations of light transport and GI are detailed in Chapter 2. Related techniques and concepts, predominantly spherical harmonics and radiosity, are described in Chapter 3.

After the foundation has been laid, the proposed algorithm is described in detail. In Chapter 4, the algorithm is derived from the previously defined concepts. Specific implementation details are given in Chapter 5. Three distinct structures for possible implementations are suggested.
Finally, the proposed algorithm is evaluated and discussed. In Chapter 6, the visual results are compared to those of other real-time GI techniques. Furthermore, the performance of the three implementations is analyzed and limitations of the proposed solution are pointed out. Chapter 7 concludes the thesis and suggests future work.
2. Global illumination foundations

In this chapter the foundations of GI techniques and light transport in general are described. Mathematical background knowledge related to finding a correct and accurate solution to GI problems is provided.

2.1 Direct lighting

As mentioned above, real-time rendering relies primarily on direct lighting which can be calculated efficiently. In this thesis, direct lighting is used as a starting point for the calculation of GI.

The specifics of direct lighting depend on the chosen representation of light sources. The following models are commonly used: directional light, point light, spot light, area light and sky light.

A *directional light source* is a light source that shoots parallel light rays in a single direction from an infinite distance. This is often used to emulate sunlight. *Point* and *spot light* are closely related. Both are infinitely small and emit light from a given position in 3D space. While a point light emits evenly in all directions, the spot light has a defined cone in which light is emitted and a given falloff function that cancels it towards the borders of this cone.

There are simple analytic solutions to shading a surface with these kinds of light sources. Shadowing could be tested by ray casting, which turns out to be prohibitively expensive for real-time rendering. Various shadowing techniques have been proposed. Shadow volumes [Cro77] work by extruding geometry from the position of the light source. Any point inside this volume is shadowed. Shadow mapping [Wil78] renders a depth texture of the scene from the position of the light source. To determine whether a surface point is shadowed, its distance to the light source is compared to the depth texture. Shadow mapping is used throughout this thesis, but the presented approach can be combined with any shadowing technique.

*Area lights* are more complex than the previous light source models. Light is not emitted from a single point in space or in a single direction, but from every point
of an area in hemispherical directions. An environment or sky light is a special case of area light. It describes a spherical or hemispherical source that represents light originating from an infinite distance in every possible direction. Handling it as a collection of area lights is a valid but expensive approach.

Area light sources can be handled in real-time [Sny96, Sch11] with various limitations. Furthermore, they produce penumbra, i.e. regions where a surface is only partly lit by an area light. Various approaches [YTD02, GBP06] for shadows of area light sources exist, but are significantly more complex than the previously mentioned shadowing techniques.

2.2 Photometric quantities

In order to understand GI and perform physically correct calculations, a number of quantities and units have to be defined and understood. Besides the photometric quantities used throughout this thesis, there are corresponding radiometric quantities. Other works may also use a slightly different notation, which makes a clear definition even more important.

2.2.1 Radiant and luminous energy

The radiant energy $Q_e$ is the total energy of the electromagnetic radiation emitted by a light source. The human eye however can only perceive light with a wavelength $\lambda$ between about 380 and 780 nm and has different sensitivities $V(\lambda)$ to different wavelengths of light. The luminous energy $Q_v$ is measured in lumen seconds and described by the equation

$$Q_v = K_m \int_{380\text{nm}}^{780\text{nm}} Q_e(\lambda) \, V(\lambda) \, d\lambda$$

(2.1)

where $K_m$ is a constant defining the relation between the radiometric unit watt and the photometric unit lumen. This equation is the core of the correlation between radiometric and photometric quantities. For the purpose of this thesis, only the visible spectrum of light is required. We avoid radiometry and rely on photometric quantities where possible.

2.2.2 Luminous flux

Luminous flux $\phi$, also known as luminous power, denotes the total luminous energy that is emitted by a light source per second. It can be described by the following equation:

$$\phi = \frac{\delta Q_v}{\delta t}$$

(2.2)

The unit of luminous energy being lumen second, it follows that the unit of luminous flux is just lumen (lm).
2.2.3 Luminous exitance

While the previous quantities describe the light source as a whole, the luminous exitance or luminous emittance $M$ describes the outgoing luminous flux per unit area:

$$ M = \frac{\delta \phi_{out}}{\delta A} \quad (2.3) $$

The unit of luminous exitance is lumen per square meter (lm/m$^2$) or lux (lx).

The radiometric equivalent is radiosity $B$. The GI technique of the same name uses this quantity extensively. To avoid confusion, the radiometric unit is used in this context.

2.2.4 Luminous intensity

$Luminous intensity$ $I$ takes the direction of the outgoing light into account. It describes the luminous flux per unit solid angle $\Omega$ emitted by a light source. The solid angle can be thought of as the three-dimensional extension of radians. It describes the area of a segment on the surface of a unit sphere, much like radians describe the length of a segment on the unit circle. Solid angle is expressed in the dimensionless unit steradian (sr). The luminous intensity is given by the following equation:

$$ I = \frac{\delta \phi}{\delta \Omega} \quad (2.4) $$

The unit of luminous intensity is candela. It is intuitively described as ‘the amount of light from a light source traveling in a given direction’.

2.2.5 Luminance

$Luminance$ $L$, for our purposes, is the most important of the quantities as it directly reveals the perceived brightness of a surface. It denotes the luminous flux $\phi$ an observer will receive when looking at a surface point from a particular angle of view $\theta$. Luminance is defined by:

$$ L = \frac{\delta^2 \phi}{\delta A \cos \theta \delta \Omega} \quad (2.5) $$

The visible surface area is proportional to the cosine of the angle between the surface normal and the light’s direction. For a human observer, the solid angle in question is the one of the eye’s pupil, i.e. the area the pupil covers on the surface of a unit sphere as seen from the surface point. The unit of luminance is candela per square meter (cd/m$^2$).
2.2.6 Illuminance

As the name suggest, *illuminance* is a quantity describing the amount of incoming light received by a surface. In contrast, all previous quantities described the amount of outgoing light. Illuminance is defined as the total incident luminous flux per unit area. To avoid confusion with the similar luminous exitance, the incident flux is denoted as \( \phi_{\text{in}} \). Illuminance is defined by the following equation:

\[
E = \frac{\delta \phi_{\text{in}}}{\delta A}
\]  

(2.6)

Just like luminous exitance described above, illuminance is measured in *lumen per square meter* (lm/m\(^2\)) or *lux* (lx).

2.3 Rendering equation

In 1986, Kajiya [Kaj86] and Immel et al. [ICG86] simultaneously introduced an integral equation known as the *rendering equation*. It models the behavior of light for the purpose of rendering, and many GI algorithms try to solve or approximate it. It can be written as:

\[
L_o(x, \vec{w}_o) = L_e(x, \vec{w}_o) + L_r(x, \vec{w}_o)
\]  

(2.7)

\( L_o \) is the total luminance, i.e. the brightness an observer in direction \( w_o \) relative to the surface would perceive when looking at surface point \( x \). \( L_e \) is the luminance directly emitted by the surface. \( L_i \) is an integral over the normal-oriented hemisphere \( \Omega \) representing the total reflected luminance from all incoming directions \( w_i \) in \( \Omega \):

\[
L_r(x, \vec{w}_o) = \int_{\Omega} L_i(x, \vec{w}_i) f_r(x, \vec{w}_i, \vec{w}_o) \langle N(x), \vec{w}_i \rangle \, d\vec{w}_i
\]  

(2.8)

\( L_i \) is the incident light at point \( x \) from direction \( \vec{w}_i \), multiplied by the *bidirectional reflectance distribution function* (BRDF) \( f_r \) described in the next section. The final term is the dot product between the normal \( N(x) \) and the direction of the incoming light. Since both vectors are normalized, this term is equal to the cosine of the angle \( \theta \) between them. Intuitively, this term accounts for the fact that light that comes from a flat angle is spread wide across the surface and thus has less influence per area. Note that this term is in range \([0,1]\) due to \( \Omega \) being a hemisphere oriented along \( N(x) \).

2.4 BRDF

The way in which incoming light is reflected is an important and complex property of a surface. The BRDF \( f_r(\vec{w}_i, \vec{w}_o) = f_r(\theta_i, \varphi_i, \theta_o, \varphi_o) \) is a 4-dimensional function describing the reflective behavior of a specific surface point. Given the two vectors \( \vec{w}_i \) for the direction of incoming light and \( \vec{w}_o \) for the direction of reflected light,
the function returns the amount of outgoing luminance in relation to the received luminance.

A BRDF can only be used to model opaque surfaces, because it is defined over the hemisphere instead of the sphere. The rendering equation can be rewritten to integrate over the full sphere to account for more complicated materials. In this case, a different surface representation than a BRDF is required. Various approaches for effects like subsurface scattering, observable in materials such as marble or wax, exist, but are beyond the scope of this thesis.

An important property of the BRDF is its eponymous bidirectionality. The underlying principle of Helmholtz reciprocity states that light proceeding from point $A$ to point $B$ can also be traced in reverse without affecting the result. This means that a light ray from a light source, being reflected, refracted, partially absorbed and eventually hitting the observer, can also be traced starting at the latter.

Although BRDFs can become arbitrarily complex, real-time computer graphics often use very simple models. An ideally matte surface is called Lambertian surface. The apparent brightness of such a surface is isotropic, i.e. independent of the observer’s angle. Furthermore, it is directly proportional to the cosine of the surface normal and the direction of incident light. This is known as Lambert’s cosine law. The BRDF can be written as the constant

$$f_r = \frac{\rho}{\pi}$$

where $\rho$ is the amount of reflected light in relation to the incoming light, known as diffuse reflectivity or albedo. Inserting Equation 2.9 into Equation 2.7 and constraining self-luminosity to an isotropic luminance yields the diffuse rendering equation

$$L_r(x) = L_e(x) + \frac{\rho}{\pi} \int_\Omega L_i(x, \vec{w}_i) \langle N(x), \vec{w}_i \rangle \, d\vec{w}_i$$

Note that the outgoing light direction $\vec{w}_o$ is no longer used in this equation. This simplification is a substantial help in the calculation of indirect illumination because the luminance emitted by a surface point $x$ is the same in every direction. In the context of GI, the luminance of point $x$ is the same for any other surface point that receives indirect light from it, independent of the direction.

## 2.5 Path tracing

Introduced along with the rendering equation in [Kaj86], path tracing is a method to find a numerical approximation to the integral $L_r(x, \vec{w}_o)$ of Equation 2.7.

Path tracing relies on Monte Carlo integration, a technique for numerical integration using random numbers. Using the law of large numbers, it can be shown that

$$\int f(x) \, dx \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$
Intuitively, an integral over some domain can be approximated by taking a finite number of random samples in said domain and averaging the results of the function to be integrated, weighted by the sample’s inverse probability. If the samples are distributed uniformly, \( p(x) \) is a constant and can be factored out, further simplifying the calculation.

To render an image with path tracing, rays are traced from the camera through the scene until they hit a light source, in accordance with the aforementioned Helmholtz reciprocity. For each path, a throughput value (initially 1) is stored. When a surface is hit at point \( x \), a reflected direction in the normal-oriented hemisphere \( \Omega_{N(x)} \) is chosen at random. The BRDF is evaluated for the respective incoming and reflected directions. The path throughput is multiplied with the result and the cosine of \( \theta \), i.e. the angle between the directions, as shown in the rendering equation. The ray is traced further until a light source is eventually hit. The light source’s luminance is then multiplied with the calculated path throughput. By averaging many of such paths per pixel, a final color value is found.

Importance sampling is a technique used to decrease the variance of the approximation. Instead of using a uniform distribution, \( p(x) \) is chosen so that random samples that are significant to the integral occur more likely. This can dramatically reduce the number of samples that have to be used to get a good approximation. The choice of \( p(x) \) depends on the integrand \( f(x) \), which in turn depends the (unknown) incoming light, the BRDF and the cosine of \( \theta \). This can be used to speed up the inherently slow path tracing algorithm considerably. For example, instead of using a uniform distribution and multiplying with the cosine, a cosine-weighted distribution can be used and the multiplication omitted. This way, less samples are generated in directions where they would have only a small influence. Some BRDFs, such as that of a perfect mirror, are not suitable to be solved with a uniform distribution. The infinitesimal direction in which the mirror reflects light would never be sampled. Thus, the distribution \( p(x) \) should be chosen carefully, in this special case as a Dirac delta function following the law of reflection.

Path tracing is an unbiased technique, meaning that no systematic error is introduced. With enough rays, and given correct surface and light source descriptions, the solution will eventually converge to the physically correct result. It is therefore often used as a ‘ground truth’ reference in computer graphics.
3. Related work

This chapter describes various approaches to approximating the rendering equation. A complete survey of GI techniques is beyond the scope of this thesis. The reader is referred to [RDGK12]. The presented techniques are similar to the solution proposed in this thesis in either their underlying principles or their goals, i.e. diffuse GI for real-time applications.

3.1 Spherical harmonics

The real spherical harmonic basis functions, in short SH, were first introduced by Pierre Simon de Laplace in 1784 [Mac67]. Based on the associated Legendre polynomials, the real SH functions form an orthonormal basis defined over the surface of the unit sphere. SH can be used to express spherical functions such as incident light from all directions for a given point \( x \). The rank \( n \) of an SH determines the number of basis functions and coefficients to be used. It is equal to \( n^2 \). A set of such coefficients defining a spherical function is also commonly referred to as SH.

While sufficiently high-ranked SH can accurately represent any spherical function, the quadratically growing number of coefficients and thus computational requirements and memory consumption may be prohibitive. On the other hand, SH with rank 2 to 4 (i.e. between 4 and 16 coefficients) are often sufficient to approximate low-frequency spherical functions as they are commonly encountered in diffuse lighting calculations.

The SH basis functions are written as \( Y_{l,m} \) where \( l \in [0, n - 1] \) and \( m \in [-l, l] \). Throughout this thesis, only rank 2 SH will be used. A definition of the first four basis functions is therefore sufficient. For a full definition of associated Legendre polynomials and SH basis functions, the reader is referred to [Gre03].

The first four SH basis functions for a Cartesian coordinate system with the y axis pointing upwards are defined as:
3. Related work

Figure 3.1: Color-coded visualization of the first few SH basis functions taken from [Gre03]. Green values are positive, red values are negative.

\[ Y_{0,0} = \sqrt{\frac{1}{4\pi}} \approx 0.2821 \quad (3.1) \]
\[ Y_{1,-1} = \sqrt{\frac{3}{4\pi}} x \approx 0.4886x \quad (3.2) \]

\[ Y_{1,0} = \sqrt{\frac{3}{4\pi}} y \approx 0.4886y \quad (3.3) \]
\[ Y_{1,1} = \sqrt{\frac{3}{4\pi}} z \approx 0.4886z \quad (3.4) \]

For brevity, the basis functions of a rank \( n \) SH are often written as \( Y_i \) where \( i \in [1, n^2] \). The mapping between the indices \( i \) and \( l, m \), as long as it is used in a consistent manner, is arbitrary. The convention used throughout this thesis is \( i = 2l + m + 1 \).

The SH basis functions form an orthonormal basis. This means that the inner product of two different basis functions is 0, while the inner product of a basis function with itself is 1. This can be expressed using the Kronecker delta \( \delta_{ij} \):

\[ \int_S Y_i Y_j = \delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \quad (3.5) \]

Due to the orthogonality of the SH basis functions, any spherical function \( f \) can easily be projected to SH coefficients. The value for each coefficient \( c_i \) (where the indices \( l, m \) were replaced by \( i \) as previously done with the basis functions) can be calculated as integral over the sphere \( S \):

\[ c_i = \int_S f(\vec{w}) \ Y_i(\vec{w}) \, d\vec{w} \quad (3.6) \]
If the function $f$ is unknown and can only be sampled (like it is often the case in lighting calculations), a numerical integration has to be used. Using the previously described Monte Carlo integration, the SH coefficients can be approximated with a finite number of samples as:

$$c_i \approx \frac{1}{N} \sum_{j=1}^{N} \frac{f(\vec{w}_j) Y_{l,m}(\vec{w}_j)}{p(\vec{w}_j)}$$ \hfill (3.7)

Reconstructing the approximated function is as simple as summing up the basis functions scaled by the SH coefficients:

$$f(\vec{w}) \approx \sum_{l=0}^{n-1} \sum_{m=-l}^{l} c_{l,m} Y_{l,m}(\vec{w}) = \sum_{i=1}^{n^2} c_i Y_i(\vec{w})$$ \hfill (3.8)

SH greatly facilitate the calculation of integrals of the sphere. Given two spherical functions $\tilde{f}(\vec{w})$ and $\tilde{g}(\vec{w})$ defined by SH with the coefficients $c_f^i$ and $c_g^i$ respectively, the integral of the product of the two functions can be written as:

$$\int_S \tilde{f}(\vec{w})\tilde{g}(\vec{w})\,d\vec{w} = \int_S \left( \sum_{i=1}^{n^2} c_f^i Y_i(\vec{w}) \right) \left( \sum_{i=1}^{n^2} c_g^i Y_i(\vec{w}) \right) \,d\vec{w}$$

$$= \sum_{i=1}^{n^2} \sum_{j=1}^{n^2} c_f^i c_g^j \int_S Y_i(\vec{w})Y_j(\vec{w}) \,d\vec{w}$$ \hfill (3.9)

Because of the aforementioned orthonormality, the integral part is equal to the Kronecker delta and equals 0 if $i \neq j$ and 1 if $i = j$. The equation can thus be simplified to only the dot product of the two coefficient vectors:

$$\int_S f(\vec{w})g(\vec{w})\,d\vec{w} = \sum_{i=1}^{n^2} c_f^i c_g^i$$ \hfill (3.10)

Introducing another function $\tilde{h}(\vec{w})$ defined by the SH coefficients $c_h^i$, the integral of the product of the three functions can be written in the same way:

$$\int_S \tilde{f}(\vec{w})\tilde{g}(\vec{w})\tilde{h}(\vec{w})\,d\vec{w} = \int_S \left( \sum_{i=1}^{n^2} c_f^i Y_i(\vec{w}) \right) \left( \sum_{i=1}^{n^2} c_g^i Y_i(\vec{w}) \right) \left( \sum_{i=1}^{n^2} c_h^i Y_i(\vec{w}) \right) \,d\vec{w}$$

$$= \sum_{i=1}^{n^2} \sum_{j=1}^{n^2} \sum_{k=1}^{n^2} c_f^i c_g^j c_h^k \int_S Y_i(\vec{w})Y_j(\vec{w})Y_k(\vec{w}) \,d\vec{w}$$ \hfill (3.11)

Unlike before, the calculation of the integral is more involved. It is detailed in Appendix A and a simple form for rank 2 SH is derived.
Another convenient property of SH is rotational invariance. Given a function \( f(x) \) and a rotated version \( g(x) \), rotating the SH projection of \( f(x) \) yields the same result as projecting \( g(x) \). Rotating SH is done using special rotation matrices. Furthermore, SH can be convoluted with any kernel function that has circular symmetry using simple multiplication. For further information, the reader is referred to [Gre03] and [Slo08].

### 3.2 Precomputed radiance transfer

The concept of precomputed radiance transfer (PRT) was introduced by Sloan et al. [SKS02], based on the work of Ramamoorthi and Hanrahan [RH01], as a technique to achieve realistic GI in real-time applications.

PRT, as the name suggests, relies on precomputation. For these computations to remain valid during runtime, the following assumptions are made:

- The scene consists of static geometry. Only the observer may be moved.
- BRDFs of all surfaces are Lambertian or close to it and cannot be changed at runtime.
- The scene is lit by an environment map that can be assumed to be at infinite distance. It can be changed at runtime.

#### 3.2.1 Overview

The luminance of a diffuse surface point \( x \) being lit by an infinitely distant light environment \( L_i \) can be deduced from the diffuse rendering equation as

\[
L_r(x) = \frac{\rho}{\pi} \int_{\Omega} L_i(x, \vec{w}_i) \left\langle N(x), \vec{w}_i \right\rangle V(x, \vec{w}_i) \, d\vec{w}_i
\]

where \( V(x, \vec{w}_i) \) is the binary visibility function. Its value is 0 if a ray from point \( x \) in direction \( \vec{w}_i \) hits any geometry and 1 if it extends into infinity. Note that this formula is valid for direct light from an environment light source only and does not take any indirect light into account.

The big number of rays required to accurately approximate the integral is prohibitively expensive under real-time constraints. However, for static scenes, the transfer function does not change. PRT makes use of this by precomputing this part of the integral.

For every vertex of a triangle mesh, a number of rays are shot in a hemisphere around the normal. The transfer function is projected onto the SH basis and stored at the vertex as \( n^2 \) coefficients, where \( n \) is the chosen SH rank.

When a ray is occluded by other geometry (i.e. \( V(x, \vec{w}_i) = 0 \)) and hits point \( y \), indirect light can still reach surface point \( x \) from \( y \). In this case, an SH is interpolated
3.3 Radiosity

Radiosity [GCT86] is a finite element approach to solving the rendering equation for diffuse scenes. The scene’s surfaces are divided into patches with approximately constant luminance. This finite number of patches is then used to perform the lighting calculations.

The eponymous radiometric quantity *radiosity* is used in this section for clarity. The corresponding photometric unit is luminous exitance.

3.3.1 Overview

Based on the assumption that the scene can be divided into patches with constant radiosity, the radiosity $B$ can be approximated by rewriting the diffuse rendering equation given in Equation 2.10 as:

$$B_j = E_j + \rho_j \sum_{i=1}^{n} B_i \cdot F_{ij}$$ (3.13)

The subscripts denote the receiving patch $j$ and the sending patch $i$ respectively. $F_{ij}$ is the *form factor*, i.e. a scalar value that determines the percentage of luminous flux that reaches patch $j$ from patch $i$. $E_j$ denotes the radiosity due to self-luminosity of patch $j$. It is not to be confused with illuminance.

The calculation of the form factor is a difficult mathematical problem. It was formulated by Lambert in 1760 [Lam60] and has been solved only in 1993 [SH93]. For a conceptually simpler solution, ray tracing can be used to approximate the form factor by testing visibility between various points on both patches.

The resulting system of equations can be solved formally as

$$B = (I - \rho F)^{-1}E$$ (3.14)

where $I$ is the identity matrix and all other symbols are the respective quantities in vector or matrix form. For a scene with $n$ patches, the size of the matrix $F$ is $n^2$. The given equation has a computational complexity of $n^3$. Both these factors quickly become prohibitive for larger scenes or higher patch resolution.

The *progressive refinement* technique [CCWG88] targets the high memory consumption by iteratively refining the result and calculating form factors on demand. The patches additionally store their *unshot radiosity*. In each step, the patch with the highest unshot radiosity is found and the $n$ form factors to all patches of the scene are calculated. The radiosity is sent and the unshot radiosity of the current sender patch is set to 0.

for position $y$ and a scaled copy is added to the SH at $x$. This way, indirect lighting can be computed without adding complexity to the runtime calculations.

PRT is applicable to glossy surfaces as described in [SKS02]. In short, different viewer angles are taken into account by calculating and storing a matrix of coefficients. Since this thesis deals with diffuse surfaces only, a full discussion is omitted at this point.
3.3.2 Real-time radiosity

The classical radiosity technique is not suited to calculate GI in large scenes under real-time constraints. However, various publications have built upon the idea of radiosity to propose methods that are considerably faster.

*Instant radiosity* [Kel97] is loosely based on the idea of radiosity. From each light source a number of random paths are traced until they hit a surface and a *virtual point light (VPL)* is created. These point lights are used to simulate indirect light reflected by the surface. This process can be repeated to simulate multiple bounces. The instant radiosity technique performs well with common graphics hardware. In combination with a shadowing technique such as shadow mapping [Wil78] or shadow volumes [Cro77], indirect illumination is correctly represented for a sufficient number of VPLs. The technique has been refined in various publications [LSK07, WFA05].

Many advanced real-time radiosity solutions have been developed in the last years. Geomerics’ *Enlighten* is a commercial middleware for computing indirect illumination. Using a point sampling of direct light across the scene’s surfaces, a radiosity solution with optional directional information is incrementally computed using a *proxy mesh*, i.e., a low-resolution representation of the scene. Enlighten has been successfully used in various game titles, such as Electronic Arts’ *Battlefield 3*. Lionhead Studios developed a real-time radiosity technique for their unreleased *Project Milo* tech demo. Similar to the technique presented in this thesis, they use a precomputed SH projection of senders to calculate illuminance at runtime. The company Molecular Matters have also developed a similar algorithm in 2012, which to this date has not been released.

Although the principles of these techniques have been briefly presented [ME10, SI11, Rei12], many details are omitted. The proposed algorithm is heavily inspired by the sparse information available. This thesis aims to fill the gaps and provide a complete overview of a real-time radiosity solution, closely related to those just mentioned.

3.4 Light propagation volumes

*Cascaded light propagation volumes (LPV)* [KD10] is a technique for approximating single-bounce indirect illumination in large, fully dynamic scenes. It can be extended to coarsely approximate multiple bounces of indirect illumination and handle participating media.

The indirect lighting in the scene is stored in one or more *light propagation volumes* in the form of SH. Initially, the volumes are filled from the surfaces causing indirect light (or low-frequency direct light, such as area light sources) using VPLs created from *reflective shadow maps* [DS05]. This technique is an extension of shadow mapping and is based on the idea that all surfaces reflecting a first bounce of indirect light from a given light source are visible in its shadow map. Each texel can be treated as an area light source. Reflective shadow maps store additional information about each texel used for indirect lighting calculations.

A volumetric approximation of the geometry is created and used for blocking light and casting indirect shadows. Light is then iteratively propagated from each cell to
its neighbor cells. The grid represents the distribution of indirect light in the scene and can be quickly evaluated as required.

LPV provide fast and stable results, if the parameters are chosen carefully. Visual plausibility is valued higher than physical correctness. The low resolution of the light propagation volume results in light bleeding, which is alleviated by the cascaded approach. A low resolution of reflective shadow maps results in flickering. Lastly, a high number of iterations has to be used to enable the propagation of indirect light over distance. Due to the low-frequency SH representation, glossy reflections cannot be handled in a reasonable manner.

The technique was implemented in Crytek’s CryEngine 3, but was replaced with a “novel real-time dynamic global illumination solution” in the successor.

3.5 Voxel cone tracing

Crassin et al. [CNS+11] introduced a GI technique known as voxel cone tracing (VCT). It approximates one or two bounces of light for fully dynamic scenes in interactive applications. VCT enables both diffuse and glossy reflections with impressive visual results.

Instead of working on the actual geometry, a hierarchical voxel representation of the scene is created as a GPU-based sparse voxel octree [CNLE09, LK10]. This representation is created once for static geometry and per frame for dynamic objects.

At runtime, the direct lighting is injected into the leaf nodes of the octree using a technique similar to reflective shadow maps [DS05]. The results are filtered and written to higher levels of the hierarchical octree (mipmapping).

To approximate the BRDF of a surface, a number of cones (approx. 5 for diffuse and 1 for glossy surfaces) are traced to calculated the illuminance received from other surfaces of the scene. Using the filtered hierarchical voxel structure, tracing is approximated efficiently by stepping along the cone and accumulating values from interpolated mipmap levels corresponding to the cone radius. This technique can also be used to calculate a good approximation of ambient occlusion.

While large scenes can be handled well using the voxel representation, cone tracing is still a costly step. Using less cones (3 diffuse, 1 specular) and a screen resolution of only 1024 × 768, the authors reported frame rates far below the real-time requirements.

A variant of voxel cone tracing known as sparse voxel octree global illumination (SVOGI) was implemented in Epic Game’s Unreal Engine 4, but taken out prior to release due to performance problems with fully dynamic lighting [Epi13]. NVIDIA’s voxel global illumination (VXGI) has been released in the form of a tech demo but can only be run using a Maxwell-powered GPU (NVIDIA GeForce GTX 960 or above) [NVI14].
3. Related work
4. Algorithm

The developed algorithm is presented in this chapter. After a brief overview, every step of the technique is described in detail. The information provided here is not dependent on the implementation, which is detailed in the next chapter.

4.1 Overview

The basic workings of the algorithm are similar to those of the radiosity technique explained in Section 3.3. During a precomputation step, the scene’s surfaces are approximated by surfels, i.e. small disk-shaped surface elements. The purpose of these surfels is to sample the direct light throughout the scene. To reduce complexity of the subsequent calculations, surfels that share similar properties are clustered into surface groups. These will represent all surfaces that diffusely reflect incoming light, similar to the sender patches in the radiosity equation (Equation 3.13).

Throughout the scene, a number of light probes are placed. These light probes receive indirect light reflected by the surface groups. In contrast to receiving patches from the radiosity technique, these light probes can not only be placed on surfaces, but at arbitrary positions to handle dynamic objects whose positions are not known a priori.

The radiosity technique uses scalar form factors to describe the mutual influence of patches. In our solution, these form factors are replaced. For every light probe, each surface group as seen from the position of the probe is projected to SH coefficients and stored. This directional information allows to shade surfaces with normal mapping and dynamic objects whose normals are unknown during precomputation.

For the shading of dynamic objects, light probes are placed in a regular 3D lightgrid. For static geometry, it is preferable to use lightmaps in order to avoid light bleeding, i.e. light being interpolated through solid objects. A lightmap is a texture that stores some kind of lighting information, in our case one light probe per pixel. To use lightmaps, a special set of UV coordinate is created for the scene. The surfaces are unwrapped in range $[0, 1]^2$ of the UV space without overlaps. The result is that every
surface point references a unique position on the texture. Traditionally, lightmaps have been used to precalculate high-frequency static lighting and save it into an RGB texture. In our approach, a set of three textures for the tristimulus values are used to store spherical SH information about low-frequency indirect lighting only. The resolution of these textures is thus considerably lower than the resolution of commonly used lightmaps.

High-frequency direct lighting often leads to problems like jagged shadows with the conventional radiosity technique. The presented approach is therefore used for diffuse indirect lighting only. At runtime, the scene and all the surfels are lit using the same common direct lighting techniques. The surfels of each surface group are averaged to calculate a single color value. The light probes then collect indirect illumination by composing a single SH from the surface groups’ color values and the SHs that have been precomputed for them. This SH can be evaluated in the direction of a surface point’s normal to get the value of indirect illumination.

Multiple bounces of light can be calculated iteratively. Surfels don’t only gather direct light, but also indirect light from the last known lightmap and lightgrid. This way, an infinite number of bounces can be simulated. Calculating e.g. one iteration per frame introduces a slight delay on subsequent bounces while keeping the computational cost low.

4.2 Precomputation

Before indirect light can be simulated, information about the static geometry is precomputed. Surfels are placed and clustered into surface groups. Light probes are distributed throughout the scene and store projected SHs for all surface groups from which they receive light.

4.2.1 Surfel placement

The goal of this step is to approximate the scene geometry by sampling a finite number of points on the surface, called surface elements or surfels. Each surfel consists of all data necessary for calculating the luminance at the surface point in question. For Lambert shading, this data is:

- **Position** $x$ in 3D space to determine whether a surfel is located in light or shadow of a light source.
- **Normal** $N(x)$ of the underlying surface, required by Lambert’s cosine law.
- **Area** $A$ to calculate the solid angle. A single surfel is interpreted as a disc on the surface.
- **Albedo** $\rho$ of the underlying material, used in Lambert’s BRDF.
- **Luminance** $L_e$ for self-luminous surfaces.
- **UV coordinates** if the underlying surface is prepared for lightmapping.
To place surfels on triangle geometry, a variety of approaches can be used. A simple method to generate unbiased, uniformly distributed samples is to sum up the total area $A_S$ of all triangles in the scene. When placing a surfel, a triangle $T$ with area $A_T$ is selected with a probability of $\frac{A_T}{A_S}$. A point $P$ in this triangle defined by the vertices $(A, B, C)$ is generated using the uniform random numbers $r_1$ and $r_2$ in range $[0, 1]$ as [OFCD02]:

$$P = (1 - \sqrt{r_1})A + \sqrt{r_1}(1 - r_2)B + \sqrt{r_1}r_2C$$  \hspace{1cm} (4.1)

The required properties for the surfel are usually stored with the vertices and can be interpolated accordingly. It is noteworthy that no high-frequency information should be used at this point to avoid aliasing effects. Albedo textures should be low-pass filtered and normal maps should be ignored during surfel distribution. Given an object $O$, that object’s total surface area $A_O$ and the number of surfels placed on that object $n_O$, the area of each surfel on $O$ is calculated as $\frac{A_O}{n_O}$.

To achieve high-quality GI results without artifacts, such as flickering indirect illumination of moving light sources, while maintaining manageable numbers of surfels, an even spatial distribution is important. A uniform distribution does not always meet this requirement as seen in Figure 4.1. To alleviate this problem, a simple iterative approach is used. After surfels have been distributed using the described algorithm, the Euclidean distance between all pairs of surfels is calculated and stored in a table. Of the two surfels that are closest to each other, one is deleted and a new surfel is placed randomly somewhere in the scene. The distance table is updated to account for the new surfel. This optimization step is repeated until the quality of the surfel distribution is sufficient.

Various other approaches could be used to create a good surfel distribution. For example, the usage of a Halton sequence instead of uniform random numbers could lead to better results to begin with. Also note that the Euclidean distance is not the best metric for the proposed optimization algorithm. To achieve an even better distribution of surfel points, the geodesic distance, i.e. the shortest distance on the mesh’s surface, could be used. Various more elaborate approaches such as [CJW+09] or [SK13] exist. However, a uniform distribution with the described optimization was found to work well enough.

### 4.2.2 Surface group clustering

The scene is now approximated by a set of surfels as a result of the previous step. To reduce the complexity of the real-time calculations, these surfels need to be clustered to surface groups. Each surface group represents a number of close surfels that are similar in their parameters. These surface groups are used to average the surfels’ luminances and calculate the resulting luminous emittance reflected by a surface, similar to a patch (in the role of a sender, not a receiver) in the radiosity technique.

Cluster analysis is the task of grouping objects into clusters based on some measure of similarity. A plethora of algorithms have been proposed for this purpose, on overview of which can be found in [JMF99].
The chosen approach is again a simple one. All surfels are initialized to be without a surface group. One surfel is chosen at random and added to a new surface group. Other surfels are then compared to all those that are already in the current surface group and added if they suffice some defined similarity criteria. For example, the Euclidean distance, normal difference and color difference can be taken into account. If optimization was used during surfel placement, the generated distance table can be used to facilitate this step.

When no more surfels can be added to the group, a new group is created and any available surfel is added at random. The algorithm is finished when all surfels are sorted into a surface group.

4.2.3 Light probe placement

*Light probes* are points in 3D space at which the illuminance due to indirect lighting and self-luminous surfaces is sampled. They store SH coefficients that are updated at runtime. By using SH instead of scalar values, directional information is stored and can be used to shade dynamic objects or surfaces with normal mapping.

At runtime, the contributing light probes for a surface point need to be found quickly. Dynamic and static objects have different requirements on how the probes are placed and on the data they contain. Therefore, light probes are kept in two separate data structures.

For dynamic objects, light probes are created in a regular three-dimensional *lightgrid*. Given the world position and normal of any point on an object’s surface, trilinear interpolation in the grid is used to find an approximate SH for the world position, which is then evaluated in the direction of the normal.

While this approach works reasonably well in some cases, the trilinear interpolation does not always produce satisfying results. Light may be interpolated through surfaces which leads to *light bleeding* or erroneous self-lighting. This is illustrated in Figure 4.2. The left light probe receives strong indirect light from the wall. The right light probe is inside and does not receive any indirect light. To shade the wall,
4.2. Precomputation

![Figure 4.2: Left: example of light bleeding taken from [KD10]. Right: illustration of erroneous self-lighting using our technique.](image)

A probe is interpolated at its position. This probe receives approximately 50% of the indirect light incident at the left probe. Thus, the wall will receive reflections ‘from itself’ on the inside. To circumvent these problems, it is preferable to use lightmaps for static scene geometry.

For each pixel of the lightmap, the world position corresponding to the pixel’s center is determined through the UV mapping and, if it is found, a light probe is created and an identifier of the probe is stored in the texture. If said UV position is not mapped to any surface, the pixel gets an invalid value. Afterwards, the probe identifiers are propagated outwards to replace these invalid values. This avoids interpolation issues along the borders of UV islands.

Various problems may arise which can be targeted by crafting the UV coordinates carefully. First, it may happen that a single pixel center is mapped to more than one surface. While this is not ideal, it is not an inherent problem for the algorithm if those surfaces are close and connected in world space. The stored low-frequency spherical information can still serve as a good approximation. The choice of world position is arbitrary. Second, interpolation can happen between surfaces that are close in UV space, but not necessarily in world space. To avoid this, there should be a gap of at least two pixels between all UV islands.

For light probes that have been created on the basis of a lightmap, the underlying surface normal is stored. This normal defines the hemisphere from which light can be received. The opposite hemisphere is blocked by the surface the light probe resides on.

Small static objects can be handled using a dedicated light probe. It is placed at the center of the object and provides spherical indirect lighting information. The object’s surface normals are then used to evaluate the lighting at runtime.

4.2.4 Light probe calculation

At the heart of the precomputation step is the projection of the surface groups visible from each light probe into SH coefficients. This is similar to the calculation of form factors in the radiosity algorithm, with the difference that spherical information are stored. Therefore, each probe does not only have information about how much light it receives from surface groups, but also directional information. The SH further roughly encodes the shape of the surface group as seen from the light probe.
To project a surface group to SH for a given light probe, each surfel is treated as a normal-oriented disc on the surface and is projected individually. Similar to Monte Carlo integration described in Section 3.1, each surfel is weighted by its solid angle, i.e. the area a surfel covers on the surface of a unit sphere as seen from the sphere’s center. Intuitively, the solid angle quantifies how large an object appears to an observer, in this case the light probe.

For each surfel \( s \) of the group, the visible solid angle in respect to the light probe \( p \) is approximated as:

\[
\Omega_{ps} = \min\left(\frac{A_s}{d^2}, 1\right) \left(\langle -N(s), \vec{w}_{ps} \rangle^+ \right) V(p, s) \text{sgn}\left(\langle N(p), \vec{w}_{ps} \rangle^+ \right)
\] (4.2)

The first term takes the surfel’s area \( A_s \) and the squared distance \( d^2 \) from light probe to surfel into account to approximate the solid angle. While this approximation works well for sufficient distances, it no longer holds for small values of \( d \). It is therefore clamped at the arbitrary value of 1. Thus, a single surfel can never cover more than 1 sr, which is about 8% of the sphere’s surface. It is better to underestimate the solid angle of a surfel and get less indirect lighting than to overestimate, which may result in artifacts and numeric overflows.

The second term describes the orientation of the surfel relative to the light probe. If the surfel’s normal \( N(s) \) and the normalized vector from light probe to surfel \( \vec{w}_{ps} \) are pointing in opposite directions, the surfel is exactly facing the light probe and the full solid angle is used. If the surfel is rotated off axis or even points away from the light probe, it is less visible or invisible. This is described by the dot product \( \langle \rangle^+ \) with negative values clamped to 0.

The third term is a visibility function between the light probe and the surfel. Because the light probe is a point in space and surfels are considered small, a single ray cast is sufficient to test visibility. The function \( V(p, s) \) returns 1 if the surfel is visible from the probe and 0 otherwise.

The fourth and final term guarantees that only surfels that are in the visible half space of a light probe are taken into account. If the light probe is not placed on a surface and thus has no normal, this term is omitted. \( \text{sgn}(x) \) returns 0 if \( x \) equals 0 and 1 if \( X \) is positive. Because \( \langle \rangle^+ \) clamps negative values to 0, these are the only two possible cases.

The SH coefficients for the projection of a surface group \( g \) in respect to light probe \( p \) are calculated as described in Section 3.1, using the following formula:

\[
c_{l,m}^{p,g} = \frac{1}{4\pi} \sum_{s \in g} \Omega_{ps} Y_{l,m}(\vec{w}_{ps})
\] (4.3)

Each of the group’s surfels \( s \in g \) is weighted by its visible solid angle \( \Omega_{ps} \). The result is normalized with the solid angle of the sphere. If any of the calculated coefficients is non-zero, the SH and an identifier of the surface group are saved in the light probe. The result is a list of surface groups significant to the light probe and an SH for each. At runtime, each precaculated SH is multiplied with the respective
4.3 Runtime calculations

Surface group’s color, yielding three SHs, i.e. one per color channel. They can then be composed into a single SH per color channel, which encodes the directional illuminance. In comparison to the projection of surface groups at runtime, this is a very cheap operation.

In addition, a sky visibility SH is calculated for each light probe. This SH encodes in which directions the sky is visible from the position of the light probe. At runtime, this data can be used to quickly evaluate an approximation of direct sky light for any given SH light environment. Without precomputed data, this is a very costly operation.

The sky visibility SH is again approximated using Monte Carlo integration. $N$ rays are shot from the light probe in directions uniformly distributed on the sphere. If the probe has a normal, a hemisphere is used instead and the number of rays is halved. The coefficients of the sky visibility SH are calculated as

$$c_{l,m}^{\text{Sky}} = \frac{4\pi}{N} \sum_{i=1}^{N} V(p, \vec{w}_i) Y_{l,m}(\vec{w}_i)$$

Since the sky visibility will not change at runtime, the values don’t need to be stored in the light probes. They can be directly written to a static lightmap or lightgrid respectively.

4.3 Runtime calculations

During runtime, the GI simulation is updated to account for lighting changes. The surfels are directly lit by all light sources and sample indirect lighting from the last known lightmap / lightgrid to simulate multiple bounces. The resulting luminances are averaged per surface group and the light probes are updated using the precalculated SHs. The changes are propagated to the lightmap / lightgrid, which are used to evaluate indirect lighting per pixel.

4.3.1 Surfel and surface group update

To update the indirect lighting simulation, the surfels are relit each frame. This step is conceptually the same as calculating direct lighting for any diffuse surface point of the scene in a real-time rendering context. All data required for a complete direct lighting calculation was gathered during precomputation.

In addition to direct lighting, the lightmap (or, in case of a surfel that has been placed on a surface that is not prepared for lightmapping, the lightgrid) from last frame is used to calculate indirect lighting. In the first frame, the lightmap is undefined, thus only direct lighting is calculated and reflected. In the second frame, a first bounce of indirect lighting is already stored in the lightmap and is used to calculate a second bounce. Consequently, light that has been reflected $n$ times is delayed by $n - 1$ frames. Because bounces for higher $n$ are less visible, this delay is hardly noticeable in practice. Evaluation of indirect lighting for surfels is again the same as for any other surface point. Details are given in Section 4.3.3.
After the luminance of all surfels has been updated, an average value $L_g$ per surface group $g$ is calculated. This way, flickering is avoided by sampling direct lighting at a high frequency, while keeping the computational cost low.

### 4.3.2 Light probe and data structure update

The input to this step is the average luminance values calculated for the surface groups. In the precomputation step, each light probe stored an SH for every surface group it can receive indirect light from. During runtime, the received illuminance from all significant surface groups is assembled into a single SH, the coefficients of which are calculated as:

$$c_{l,m}^p = \sum_{g \in G} L_g c_{l,m}^{p,g}$$

This equation neglects the color of the light. In theory, the calculation would have to be performed for each visible wavelength of light. In practice, this is reduced to calculations for red, green and blue. This means that three SHs are created in this step, one for each color channel.

Figure 4.3 shows the assembled illuminance SHs (averaged over the three channels for the shape) of the lightmap. These SHs visualize the direction and strength of received indirect light.

The assembled indirect light SHs of the lightgrid’s light probes can be written directly to the data structures. In the lightmap, a single light probe may be used multiple times, especially along the borders of UV islands. These probes are therefore calculated individually and the SHs are copied to the lightmap afterwards.
4.3. Runtime calculations

4.3.3 Evaluating the SH

To evaluate indirect lighting for any given surface point of the scene, first the corresponding SHs are determined. If the surface point has lightmap UV coordinates, the bilinearly interpolated SHs for indirect light (red, green and blue) and sky visibility are read from the lightmap. Otherwise, the three-dimensional lightgrid is consulted and the values are read using trilinear interpolation.

The diffuse rendering equation given in Equation 2.10 can now be rewritten and approximated using the illuminance from the SH:

$$\mathbf{L}_r(\mathbf{x}) \approx \frac{\rho}{\pi} \int_{\Omega} \left[ \sum_{j=1}^{n^2} c_j Y_j(\vec{w}_i) \right] \langle N(\mathbf{x}), \vec{w}_i \rangle \, d\vec{w}_i \quad (4.6)$$

To approximate the computationally expensive integral, the SH can be convolved with a cosine kernel and later evaluated in the direction of the surface normal $N(\mathbf{x})$ [Slo08]. The convolution is done by simply scaling the SH coefficients with precomputed factors. For rank 2 SH, these factors are $\{\pi, \frac{2}{3}\pi, \frac{2}{3}\pi, \frac{2}{3}\pi\}$.

While this approach is sensible for SH of higher ranks, we use low-rank SH and thus already have a low-frequency approximation of the illuminance. Convolution would further blur the information and decrease directionality. This can also be seen in the factors, which weight the first, omnidirectional coefficient higher than the directional ones. To preserve as much directional information as possible, a constant factor of $\pi$ was chosen instead:

$$\mathbf{L}_r(\mathbf{x}) \approx \frac{\rho}{\pi} \sum_{i=1}^{n^2} \pi c_i Y_i(N(\mathbf{x})) = \rho \sum_{i=1}^{n^2} c_i Y_i(N(\mathbf{x})) \quad (4.7)$$

The evaluation of the sky light involves three components: a cosine lobe over the normal $N(\mathbf{x})$ defined by $c_i^{\cos}$ to weight the incoming light, the precomputed sky visibility SH defined by $c_i^{\text{vis}}$ and the sky SH defined by $c_i^{\text{sky}}$. The sky SH is actually a set of three SHs, one for each color channel, which are treated equally. The colors are therefore omitted in the formula.

The SH for the cosine is easily constructed from the normal [Day10] as:

$$c_1^{\cos} = \sqrt{\frac{\pi}{2}} \quad c_2^{\cos} = \sqrt{\frac{3}{\pi}} N_x(\mathbf{x}) \quad c_3^{\cos} = \sqrt{\frac{3}{\pi}} N_y(\mathbf{x}) \quad c_4^{\cos} = \sqrt{\frac{3}{\pi}} N_z(\mathbf{x}) \quad (4.8)$$

To solve the sky visibility, the integral of the product of the three components is calculated. The derivation is given in Appendix A. It can be written as:

$$\int_S \left( \sum_{i=1}^{n^2} c_i^{\cos} Y_i(\vec{w}) \right) \left( \sum_{i=1}^{n^2} c_i^{\text{vis}} Y_i(\vec{w}) \right) \left( \sum_{i=1}^{n^2} c_i^{\cos} Y_i(\vec{w}) \right) \, d\vec{w} = -\frac{1}{2\sqrt{\pi}} \sum_{i=1}^{4} \rho c_i^{\text{sky}} \quad (4.9)$$
The new SH coefficients $c_p^i$ are calculated as:

$$c_1^p = c_2^\cos \cdot c_4^\cos - c_3^\cos \cdot c_3^\cos + c_4^\cos \cdot c_2^\cos - c_1^\cos \cdot c_1^\cos$$

$$c_2^p = c_1^\cos \cdot c_4^\cos + c_4^\cos \cdot c_1^\cos$$

$$c_3^p = -c_4^\cos \cdot c_3^\cos - c_3^\cos \cdot c_1^\cos$$

$$c_4^p = c_1^\cos \cdot c_2^\cos + c_2^\cos \cdot c_1^\cos$$

Note that this new SH only depends on the cosine and the sky visibility and can be combined with all three color channels of the sky SH without the need for recalculation.
5. Implementation

This chapter provides various details on implementing the described algorithm. A prototype was created in C++ using Microsoft DirectX 11. Example application code is thus given in C++, shader code in High Level Shading Language (HLSL).

5.1 Precomputation implementation

This section goes into detail on the implementation of the precomputation steps described in Section 4.2.

5.1.1 Surfel placement

For each generated surfel, barycentric coordinates are generated using the random distribution and optimization described in Section 4.2. Surfel data is interpolated using the generated coordinates and stored in the following data structure:

```c
struct Surfel {
    float3 position;
    float3 normal;
    float3 albedo;
    float2 lightmap_uv;
}
```

Listing 5.1: Surfel data structure

In contrast to Section 4.2.1, two values have been omitted. The luminance $L_e$ is not stored because there are no self-luminous surfaces in our test scene. More important, the area of a surfel is not stored. The surfel density is the same in the entire test scene, with the result that each surfel represents the same area. This value is stored globally. If objects could have varying surfel densities, which may well be the case in a practical application, this value could not be omitted.
While linear interpolation of world position and UV coordinates works without any problems, normal and albedo are potentially error-prone. The normal needs to be renormalized after interpolation to preserve unit length. A precaution should be taken for the rare case that the result of the interpolation is a zero vector, in which case a renormalization would result in a division by zero.

The subsequent calculations in theory require a perceptually linear color space such as the Lab color space. The albedo is stored as a linear RGB color which, although not perfectly linear, is sufficient to deliver convincing results.

In practice, the color values for albedo interpolation are usually taken from textures, using a low mip-map level to avoid aliasing due to high frequencies. These textures are commonly stored in sRGB format, which can be used to directly display images, but is not suitable calculations because it is not linear. The read color values have to be relinearized. This is done by applying a gamma correction with the commonly used value of $\gamma = 2.2$:

$$c_{\text{linear}} \approx c_{\text{srgb}}^{2.2}$$

The final pixel color values need to be converted to sRGB to be displayed correctly on common computer monitors. Thus, a gamma correction with the inverse $\gamma = \frac{1}{2.2}$ is applied as a last step before displaying the calculated color values. Further information on the importance of doing calculations in a linear color space can be found in [Gd08].

### 5.1.2 Surface group clustering

Surfels are clustered as described in Section 4.2.2 to form surface groups. A surface group is initially stored as a dynamic array of surfel pointers. After all surface groups have been generated, it is advisable to relocate the surfels to be stored in a single consecutive memory block. Inside this block, each surface group’s surfels are stored in sequence. When averaging a surface group, all surfels in the respective group need to be accessed. A consecutive memory layout minimizes cache misses and thus the time required to finish the operation.

When all surfels are stored in a block of memory, a surface group no longer needs to store an explicit list of surfels. It is sufficient to store the offset of the first surfel and the surfel count of this group. This is illustrated in Listing 5.2.

```c
// Consecutive block of memory
Surfel* g_surfels;

struct SurfaceGroup
{
    unsigned int surfelOffset;
    unsigned int surfelCount;
}
```

Listing 5.2: Optimized surface group data structure
During the process of clustering surfels into surface groups, the *difference* between two surfels has to be determined in order to decide whether a surfel should be added or not. There are many different ways to implement this function. A practical example using the surfel position and normal is given in Listing 5.3.

```cpp
static bool isSurfelAcceptable(const Surfel & x, const Surfel & y, float threshold) {
  float totalError = 0.0f;

  // Position difference: squared euclidean distance (cm^2)
  float3 distError = x.position - y.position;
  totalError = dot(distError, distError) / 15.0f;

  // Normal difference based on angle
  // Opposite normals result in a very high error
  float dotP = max(dot(x.normal, y.normal), 0);
  float normalDifference = (1 - dotP) * 10000.0f;
  totalError += normalDifference;

  return totalError <= threshold;
}
```

Listing 5.3: Example code for the surfel difference function, to be used with a threshold of approximately 1500. The surfel color is not considered.

### 5.1.3 Light probe placement

In this step, positions and, if applicable, normals of the light probes are determined and stored in a temporary data structure. This information is only required during precomputation and will be discarded afterwards.

The placement of light probes in a regular grid is trivial. The normals of those probes are set to the zero vector.

Because a single probe can be used multiple times in a lightmap, e.g. when duplicated along the borders of UV islands, lightmaps store indices into a global list of probes. To create light probes for the lightmap, we initialize it with invalid probe index values and loop over all triangles of the scene. The lightmap UV coordinates are read and each invalid pixel’s center is tested against the triangle. If the pixel is contained, the world position and normal are interpolated and a light probe is created. At the same time, the index of the newly created probe is stored in the lightmap.

### 5.1.4 Light probe calculation

Both major steps during light probe calculation, i.e. the projection of surface groups to SH and the calculation of the sky visibility SH, rely on visibility test using ray tracing. For the prototype the Intel Embree library [Int14] has been chosen as a fast and flexible ray tracing solution.

To make the computations efficient, it is advisable to save ray casts where possible. If a surfel is not in the positive hemisphere (in case of a light probe with a normal) or
points away from the light probe, it is invisible and no ray has to be cast. These tests can be done using simple dot products. Also note that the costly normalization of the direction vector between light probe and surfel can be delayed after these tests, because only the sign of the dot product is taken into account.

Similar to the relation between surface groups and surfels before, information about the projections are stored in an optimized data structure:

```c
struct ProjectionInfo
{
    unsigned int surfaceGroupIndex;
    float4 solidAngleSh;
}

// Consecutive block of memory
ProjectionInfo* g_projectedSurfaceGroups;

struct LightProbe
{
    float4 skyVisibilitySh;
    unsigned int projectedSurfaceGroupOffset;
    unsigned int projectedSurfaceGroupCount;
}
```

Listing 5.4: Optimized and calculated light probe data structures

When composing the three SHs of a light probe (one per color channel) from the significant surface groups, this data structure ensures that a minimum of cache misses occur when fetching the projected surface group information.

Depending on memory constraints and target quality, it may be advisable to not store all non-zero SHs and surface groups, but only the \(n\) most significant. To sort the SHs by *significance*, some heuristic such as the magnitude of the coefficient vector is used. Instead of using a fixed maximum number, it is also possible to include SHs whose significance exceeds a certain threshold value.

### 5.2 Runtime implementation

Contemporary PC and game console hardware features two prominent processing units. The *central processing unit (CPU)* is the main general purpose processor. It is typically fast and, although multi-core processors allow for some parallelism, suited best for tasks that need to be carried out in serial. The *CPU* has direct access to the machine's *random access memory (RAM)* or main memory, which has become a plentiful resource in the last years.

The second one is the *graphics processing unit (GPU)* which is heavily optimized for the rasterization of triangles and massively parallel processing of individual pixels. The GPU can be used as a parallel processor for a variety of tasks by reshaping the problem at hand to fit the traditional rasterization pipeline. In addition, technologies like Compute Shaders or NVIDIA’s *Compute Unified Device Architecture (CUDA)* have been specifically designed to provide easy access to a GPU’s parallel processing
5.2. Runtime implementation

In contrast to the CPU, the GPU doesn’t have access to the main memory. It features its own video memory that is typically smaller than the main memory and can only be accessed for reading or writing during specific stages of the rasterization pipeline.

This section provides an overview of various possible implementation structures using the described hardware. A performance test and in-depth comparison can be found in Section 6.2.

5.2.1 Pure CPU solution

![Data Flow Diagram]

Figure 5.1: Data flow of the pure CPU solution

The first way to implement the algorithm is to rely on the CPU as much as possible. All calculations, from the relighting of the surfels to the update of the data structures, are done on the CPU. To easily integrate the evaluation of indirect light into an existing GPU-based renderer, the lightmap and lightgrid are transferred from RAM to video memory each frame. The GPU’s pixel shader then takes care of applying the indirect lighting to each pixel of the final rendered image by reading from these data structures.

Surfel and surface group update

Every surfel belongs to exactly one group. To calculate the average luminance of a surface group, the surfel luminance values can be accumulated directly. There is no need to even temporarily store the luminance values of all surfels in the scene. Each group’s surfels should be stored consecutively in memory. This minimizes cache misses and leads to better performance.

To light a surfel, each light source is considered independently and the resulting luminances are accumulated for the surfel. How this is done specifically depends on the type of the light source, as mentioned in Section 2.1. To determine shadowing, the visibility between a surfel and a light source needs to be tested. For directional lights, point lights and spot lights, a single ray is cast from the center of the surfel to the position of the light source. For area lights, numerous rays would need to be cast, which quickly renders shadowed area lights impractical. Sky lights use the precomputed sky visibility which is very fast but does not provide shadowing from dynamic objects.

In addition to accumulating direct light from the scene’s light sources, indirect light from the previous frame is taken into account to simulate multiple indirect bounces iteratively with each frame. Values can be read from the previous lightmap / light-grid which are still stored in RAM, but need to be interpolated by hand.
Light probe and data structure update

After the average luminances of the surface groups have been stored in RAM, the spherical indirect lighting information is assembled in the light probes. This step is straightforward: loop over the light probe’s significant surface groups, multiply the respective precomputed SH with the surface group’s luminance yielding three SHs (red, green and blue) and accumulate the results.

While the significant surface group IDs and their precomputed visibility SHs can be laid out consecutively in memory, the access to the surface groups’ luminances cannot be predicted. This results in a lot of cache misses that are not easily avoided. The prototype code was optimized using Streaming SIMD Extensions (SSE) instructions, which proved to be the most efficient method. A multi-threaded solution resulted in even more cache misses, further slowing down the application. Fused Multiply-Add (FMA) instructions may accelerate this step further, but are not yet widely supported and therefore weren’t tested.

After the light probes have been updated, the data structures are sent to the GPU and stored in textures (lightmap) and volume textures (lightgrid) in video memory. This transfer is an inherently slow operation.

5.2.2 Hybrid solution

The pure CPU solution has various drawbacks, the first of which is the relighting of the surfels. As stated in Section 4.3.1, the process of lighting surfels is the same as lighting any diffuse surface point in the scene. Accumulating light from various light sources and simulating shadowing on the GPU are well understood fundamentals of computer graphics that are most probably already implemented in an existing real-time rendering context. It is therefore sensible to leverage these techniques. In the previous solution, a representation of the scene optimized for ray casting had to be kept in RAM during runtime. The ray casts themselves are a slow operation, which limits the number of surfels and especially light sources that can be used with the pure CPU approach. Relighting surfels on the GPU negates this overhead by using solutions that are already in place.

Surfel and surface group update

The idea behind the relighting of surfels on the GPU is borrowed from deferred rendering [DWS+88, ST90]. Using this technique, rendering of a scene is in general done in two passes. In the first pass, the scene geometry is rasterized and all information required for shading a single pixel is stored into a set of textures known as the G-buffer. This information include e.g. diffuse albedo, surface normal, world
position and UV coordinates. The second, deferred pass rasterizes only a single triangle filling the whole viewport. For each pixel of the render target, information is read from the G-buffer and used to calculate the final color value.

This idea is adapted to relight surfels on the GPU. A G-buffer is created that stores information about one surfel per pixel. Since the precomputed surfel data does not change at runtime, the surfel G-buffer is read-only and initialized when the scene is loaded. This also means that to relight the surfels, only a deferred pass is needed, since the surfel G-buffer is already in place. The size of the textures is arbitrary as long as it can store the data of all surfels. An example surfel G-buffer is shown in Figure 5.3.

The information required to relight a surfel follows from Listing 5.1. The data can be stored in 11 float values. In our implementation, 3 textures with 4 float channels each (DirectX format DXGI_FORMAT_R32G32B32A32_FLOAT) are used, where position, normal and albedo are stored in RGB channels and lightmap UVs are stored in two of the alpha channels. The remaining alpha channel could e.g. be used to store a scalar value which defines the self-luminance of the surfel relative to the albedo, allowing for surfaces to glow in their albedo color. In our implementation, this value is unused.

It is noteworthy that even if surfels are allowed to have different areas (which is not the case in the prototype implementation), this value is not needed at runtime. It is only used during the projection of surface groups into SH and implicitly encoded in the results.

If the technique is implemented into a context that already features a deferred renderer, a large part of the deferred pixel shader can be reused. For each surfel, the luminance due to each light source is calculated and the values are accumulated. In contrast to the CPU solution that relies on expensive ray casting operations to determine shadows, GPU-based surfel relighting can use existing shadowing techniques that are already used for direct lighting. As mentioned in Section 2.1, shadow maps are used in our implementation.
After the surfels have been lit, the luminance values are transferred from video memory to RAM. The CPU then, instead of calculating the luminances, just reads them from memory and averages the stored values.

**Light probe and data structure update**

As a result of the previous step, the average luminances of the surface groups are again stored in RAM. Therefore, this step is exactly the same as with the pure CPU solution and does not need to be adapted.

### 5.2.3 Pure GPU solution

The hybrid solution makes surfel relighting more manageable and fast. The greater part of the frame time is due to two reasons which could be alleviated by relocating the whole workload to the GPU. First, the light probe update is memory-bound. GPUs usually possess a greater memory bandwidth than CPUs, which indicates that this step could run faster on the GPU. Second, the transfer of surfel luminances from the video memory to RAM and the transfer of the lightmap and lightgrid back to video memory are slow and could be avoided by using a purely GPU-based solution.

**Surfel and surface group update**

Using the surfel relighting from the hybrid solution, the surfel luminances are stored in an intermediate texture. This texture can directly be used to calculate the average luminances of the surface groups.

The surface group update requires another render target and a G-buffer storing the metadata of the surface groups defined by Listing 5.2. The surface group G-buffer is created along with the surfel G-buffer during precomputation. To save space and bandwidth, the surfel offset and count can be encoded into a single 32 bit integer (DirectX format `DXGI_FORMAT_R32_UINT`), with 24 bits for the offset and 8 bits for the count. This sets an artificial limit of 256 surfels per surface group and about 16 million surfels in total, both of which should be more than enough for any scenario.

To update the surface group luminances, a deferred pass is used to write to the surface group luminance render target. The pixel shader first loads the respective surface group metadata. The 24 bit surfel offset and 8 bit count are extracted from the 32 bit value. Using these values, a loop over the surface group’s surfels accumulates the values before dividing by the surfel count to return the average. Example shader code is given in Listing 5.5.
5.2. Runtime implementation

```c
SamplerState g_clampSampler : register(s0);
SamplerState g_clampPointSampler : register(s1);
Texture2D<guint> g_surfaceGroupData : register(t0);
Texture2D g_surfelLuminances : register(t1);

float4 main(VS_OUTPUT f) : SV_TARGET
{
    float3 result = float3(0, 0, 0);
    uint width, height;
    g_groupBuffer.GetDimensions(width, height);

    int3 groupPos = int3(f.uv.x * width, f.uv.y * height, 0)
    uint metadata = g_surfaceGroupData.Load(groupPos);
    uint offset = metadata % 16777216;
    uint count = (metadata - offset) / 16777216;

    g_surfelLuminances.GetDimensions(width, height);
    for (uint i = offset; i < offset + count; ++i)
    {
        float3 pos = float3(i % width, i / width, 0);
        result += g_surfelLuminances.Load(pos).xyz;
    }
    result /= count;

    return float4(result, 1);
}
```

Listing 5.5: HLSL pixel shader code for surface group update

Light probe and data structure update

The update of light probes is conceptually very similar to the averaging of surface groups. This time, three render targets for the red, green and blue SH coefficients are required. During initialization, the projected surface group data and light probe metadata are written to a total of three static textures: the indices of the projected surface groups are written to a 32 bit integer texture, their corresponding SHs to an RGBA floating point texture; the light probe metadata is written to a 32 bit integer texture in the same way as the surface group metadata before. Furthermore, the light probe indices of the lightmap are written to a texture during initialization.

Schematically, the update of a single light probe on the GPU works like this:

- Read light probe metadata to determine which projections are used.
- Loop over all the projections and for every projection:
  - Get the SH of the projection.
  - Get the surface group index of the projection.
  - Using the surface group index, get the average luminance of the surface group (calculated in the previous step and stored in an intermediate render target).
  - Add the product of the SH and the luminance to the result SHs.
The pixel shader starts by reading the light probe metadata and extracting the projected surface group offset and count. Just like the previous shader, a loop using these values then accumulates the final result. In contrast to the surfel group averaging, another indirection is taken during light probe update. The projected surface group index and corresponding SH are read. After that, the average luminance of the surface group is read using the previously acquired index. The projected surface group’s SH is then multiplied with each color channel of the luminance, yielding three SH which are then added to their respective result SH.

The update of the lightmap is a simple copy operation. Light probes are read according to the index texture and the SHs are written to the red, green and blue lightmaps. The lightgrid’s light probes can be calculated directly and written to the volume texture one slice at a time, which can be done using multiple draw calls or a geometry shader.
6. Evaluation

In this chapter, the results of the proposed algorithm are presented and compared to the results of related techniques. The performance of the suggested implementations is assessed and limitations inherent to the technique are shown. The popular Crytek Sponza scene, available at http://graphics.cs.williams.edu/data/meshes.xml, was used for all images and tests.

6.1 Results

In Figure 6.1, the results of light propagation volumes (LPV), voxel cone tracing (VCT) and our approach are compared to a path tracing reference image. Due to the iterative propagation approach, LPV is not able to produce distant indirect light. Given the limited number of bounces supported by this technique, no convincing ambient light is produced, while short range indirect light is exaggerated. The same behavior can be observed for VCT, although somewhat less severe. In contrast, our technique features balanced indirect lighting over all distances. Thanks to a high number of bounces, ambient light is scattered all throughout the scene, realistically illuminating e.g. the colored cloth on the opposite side.

The low-order SH store only a coarse approximation of directional information. Sudden lighting changes on edges, as seen on the ceiling in the reference image, are only possible if the lightmapping is crafted carefully, i.e. if the adjacent surfaces are disconnected in UV space. This can be seen on the pillar in the foreground. VCT handles this slightly better, but still produces blurry results in comparison with the reference.

Figure 6.2 shows another comparison between LPV and our approach. The LPV image was created using NVIDIA’s implementation from the DirectX 11 SDK, available online at http://developer.nvidia.com. The settings were chosen to balance the aforementioned discrepancy between short and long range indirect illumination as much as possible. Using 32 iterations and a second light bounce, which already results in frame times beyond the given real-time constraints, does not help to produce convincing ambient light. Turning up the indirect light leads to strongly overexposed
short range indirect illumination. Using the same direct light and no sky light or ambient term, our approach produces realistic and appealing lighting in the entire scene. It seems clear that LPV was created as an enhancement and not as a full diffuse GI solution.

An example of a dynamic object being shaded in a scene with a strong directional light and a dim sky light can be seen in Figure 6.3. The object is shown from two different camera angles and has not moved in between. Notice the subtle color bleeding from the cloth hanging off the wall. Dynamic objects can be moved without flickering or temporal artifacts, but are not perfectly handled by our algorithm. Limitations are discussed in Section 6.3.

Figure 6.4 shows the scene lit entirely by a single spot light pointed straight at the wall on the left. All lighting but the direct light in the clearly visible aperture angle is calculated in real-time using the proposed algorithm. The light can be freely moved or rotated. Lighting a big scene with a single bright light source is a potentially problematic edge case. Details are again given in Section 6.3.

### 6.2 Performance

The precomputation step, depending on the chosen quality settings, takes anywhere from a few seconds to various minutes. Since of the focus of this thesis is on the
Figure 6.2: Comparison of ambient light with light propagation volumes (top) and our approach (bottom).
6. Evaluation

Figure 6.3: A dynamic object (same world position, two different camera angles) receiving colored indirect light using our approach.

Figure 6.4: Test scene lit using a single spot light. The only direct light emitted by the spot light is visible as a bright circle on the left wall. All other light is indirect and calculated using our approach.
6.2. Performance

runtime computations, the precomputation step is not fully optimized in our implementation. Thus, no detailed timings are reported at this point.

All three suggested runtime implementations were thoroughly tested on a PC with an Intel Core i5-3450 CPU and an AMD Radeon HD 7770 GPU. It will be referred to as test system 1. For comparison, a quick test was performed on a second system with an AMD Phenom II X4 955 CPU and an AMD Radeon HD 7850 GPU, referred to as test system 2. Unfortunately, the system was not available long enough to record extensive timings. The purpose of this second system is therefore not to provide any additional data, but to give an impression of the algorithm’s performance across different systems.

Note that even though the described algorithm may run faster using one implementation over another, the computation times depend on many factors. Different hardware configurations may lead to vastly different results. Optimizing memory access patterns is often more important than optimizing the actual calculations. Also, if the algorithm is built into a given environment, e.g. a game engine, it is crucial to spread the workload across CPU and GPU as evenly as possible. For example, if the processing time of a frame is already bound by the GPU, meaning that the CPU idles while waiting for results from the GPU, a (partly) CPU-based implementation may be preferable.

6.2.1 Computation time

Table 6.1 shows the computation time of the algorithm on test system 1 for each of the suggested implementations and various settings. The measured timings include surfel relighting, surface group averaging, light probe and data structure update, and data transfer between CPU and GPU as far as required by the respective implementation. For simplicity, only lightmaps were used and the lightgrid (which works in the same way) was excluded. The number of projected surface groups per light probe was limited to 256. Pixel shading, i.e. querying of the data structures and application of sky lighting, is not included. For a screen resolution of 1920×1080 pixels, this step takes about 1.5 ms for all settings shown.

The table shows the chosen number of surfels, the resulting number of surface groups after clustering, the chosen lightmap pixel resolution and the resulting number of generated light probes. The number of surfels was doubled in each step. It is noteworthy that this usually does not result in twice as many surface groups because they are roughly constant in size. The resolution of the (square) lightmap was also doubled in each step, resulting in approximately four times as many light probes. For simplicity, a lightgrid was not used. It is handled in essentially the same way as the lightmap and thus has the same effect on the computation time. For the chosen test scene, 65536 surfels and a lightmap resolution of 128² produces high quality results.

For the CPU implementation, there is a clear correlation between the number of surfels and the measured time. The ray casts required to relight surfels on the CPU quickly become too expensive for real-time rendering. This step becomes even slower for bigger scenes than the Sponza test scene, because more geometry has to be traversed during ray casting. The light probe update does not have as much
impact on the total computation time. However, the implementation of this step is optimized using SSE instructions and could not be sped up further using multithreading which suggests that this step is bound by memory bandwidth. Considering the large number of unpredictable and random memory access operations, this does not come as a surprise.

On test system 1, the hybrid implementation performs best. It scales gracefully with both the number of surfels and the number of light probes and stays within real-time constraints for all tested settings. On test system 2 however, this implementation runs a lot slower and is beaten by the GPU implementation. The system features a slightly weaker CPU and slightly stronger GPU, which contributes to this observation. Furthermore, SSE instructions, being designed by Intel, seem to be more optimized on the Intel CPU. It is possible that, using a different optimization strategy, this implementation could be sped up on the AMD CPU.

While the surfel relighting step is again unproblematic, the GPU implementation struggles with light probe updating on test system 1. This was handled slightly better on test system 2, probably due to the GPU’s superior memory bandwidth (153.6 GB/s vs. 72 GB/s). The reason why the GPU solution is so much slower than the hybrid solution is hard to see. We suspect that the caching strategies of the CPU, which are well understood and easy to optimize for, harmonize better with our algorithm than those of the GPU, which are not fully revealed.

As mentioned above, the reported time measurements depend on many factors and should be used as a guideline. To achieve the best possible performance for a given application context and target hardware, different implementations and optimization strategies should be considered.

### 6.2.2 Memory consumption

One advantage of our technique is the low memory consumption compared to other GI techniques. As stated in [CNS+11], VCT uses about 1024 mb of video memory and LPV uses even more for comparable settings.
6.3 Limitations

The precomputed data of each surfel is stored in twelve 32 bit float values, resulting in a size of 48 bytes per surfel. For 65536 surfels, this amounts to 3 mb of memory. The memory footprint of surface groups, being few in number and stored in a mere 32 bits each, is negligible.

The majority of the required memory is consumed by the precomputed light probe data. Each probe stores a sky visibility SH in four 32 bit float values, i.e. 16 bytes. In addition, a 16 byte SH and a 2 byte surface group index are stored for each projected surface group. Note that even though a maximum number of 256 projected surface groups per light probe were allowed in our tests, many probes are influenced by less surface groups than this. For 5182 light probes this results in an average number of 166 significant surface groups per probe and about 14.8 mb of memory consumption.

The dynamically updated lightmaps and the textures / memory blocks for intermediate results add another few megabytes. For the highest settings tested, the memory consumption totals at around 100 mb. Note that depending on the chosen implementation, data either resides in video memory, main memory or partly in both. This should also be taken into account when considering the choice of implementation structure.

6.3 Limitations

Although the presented technique performs well and produces visually pleasing results in most cases, it does not come without problems and limitations.

Our algorithm treats all surfaces as purely diffuse. There is currently no way to easily include glossy reflections in the calculations.

The treatment of dynamic objects is limited. Since they are not considered during precomputation, they do neither reflect light nor cast indirect shadows. Using the lightgrid, dynamic objects are subject to light bleeding, as described in Section 4.2.3.

The light probes of the lightgrid show a problem inherent in the chosen illuminance representation. An SH representing very strong light coming from one direction inevitably encodes a ‘negative light’ from the opposite direction. Adding up these SHs can result in dark areas or wrong colors, as seen in Figure 6.5 (1). This edge case is alleviated in more balanced light situations, e.g. when using a sky light, as seen in Figure 6.5 (2).

Indirect shadows from static geometry are subject to the same problems as shadows in the classic radiosity technique. Due to the low resolution of the lightmaps, indirect shadows appear jagged as seen in Figure 6.5 (3). They may also be disconnected from the shadow caster or slightly leak through geometry.

Limiting the number of projected surface groups per light probe is a convenient way to increase performance and use the algorithm in larger scenes. The surface groups are sorted by significance, i.e. the magnitude of their projected SH, and only the $n$ most significant are taken into account. However, if a seemingly less significant surface group is lit very brightly, it may still have a visible impact on the light probe. This is illustrated in Figure 6.5 (2), where only the most significant 64 surface groups
6. Evaluation

Figure 6.5: Problems with the presented technique. (1): Wrong colors resulting from strong light in the opposite direction. (2): An added sky light alleviates the problem. (3): Jagged indirect shadows from a strong point light source. (4): Dark spots resulting from a low number of surface groups per light probe used in a difficult light situation.

were stored per light probe. The whole scene is lit by a single very bright point light behind the camera. For some of the distant light probes, the surface groups close to the camera were not considered significant. These bright surface groups are therefore not taken into account, resulting in dark spots on the floor.

Ambient occlusion is usually too high-frequency to be captured by our approach. It is easily possible to combine it with a technique such as screen-space ambient occlusion [BS09] to approximates the effect.

More practical problems may arise when implementing the algorithm into an existing application. For example, shadow maps are often adjusted to the camera frustum to avoid perspective aliasing and make optimal use of the available pixels. This means that there is no shadow information available outside the frustum, e.g. behind the camera. However, surfel relighting requires this information, since surface groups behind the camera may well have an effect on light probes in the frustum and thus the rendered image.
7. Conclusion

In this thesis, a fast, robust and flexible approach for the calculation of diffuse GI under real-time constraints was presented. Light sources can be moved freely at runtime. Static geometry is handled very well using a precomputation step. Dynamic objects are subject to the described limitations, but can be shaded and don’t show temporal artifacts.

Our approach simulates an infinite number of light bounces. In contrast to other GI techniques, the computational cost is not dependent on the distance between sender and receiver of indirect light. The presented algorithm is therefore suitable for creating convincing ambient light. Large parts of a scene can be lit with only a single light source, instead of relying on a constant ambient term or manually placing fill lights.

7.1 Future Work

Some parts of the algorithm have not been examined in detail. For example, the placement of surfels and clustering of surface groups are well studied problems as described in Section 4.2.1 and Section 4.2.2. Simple approaches were used to not go beyond the scope of the thesis. However, the algorithm may be improved by using more sophisticated techniques during precomputation.

The handling of dynamic objects could be improved. Projecting a coarse approximation of a dynamic object to SH for all (close) light probes could be sufficient to implement indirect reflections from dynamic objects, although this may be a too costly operation. Using this SH to ‘mask out’ surface groups that are blocked by the object, the calculation of dynamic indirect shadows could be facilitated.

The used rank 2 SH are very low-frequency. It is to be evaluated whether higher-rank SH could improve the results. For light probes placed on surfaces, hemispherical harmonics [HZS+06] are sufficient and could improve the quality and efficiency of the technique. Also, an entirely different basis such as the hemispherically orthonormal $H$-basis [HW10] could be used instead of SH.
Currently, the technique works only for diffuse surfaces. Further research is necessary to find a solution for glossy reflections. As a starting point, the existing SHs may be usable. Light probes store directional illuminance, in contrast to patches in the classic radiosity technique which store isotropic luminance. This could be leveraged to calculate a single glossy bounce of indirect light or, in combination with a higher-frequency representation, even render glossy reflections, which would greatly increase the versatility and visual quality of the presented approach.
Appendix A

As stated in Section 3.1, the integral of the product of the three spherical functions \( \tilde{f}(\vec{w}), \tilde{g}(\vec{w}) \) and \( \tilde{h}(\vec{w}) \) defined by SH coefficients \( c^f_{l,m}, c^g_{l,m} \) and \( c^h_{l,m} \) respectively can be written as:

\[
\int_S \tilde{f}(\vec{w})\tilde{g}(\vec{w})\tilde{h}(\vec{w}) \, d\vec{w} = \int_S \left( \sum_{lm} c^f_{l,m} Y_{l,m}(\vec{w}) \right) \left( \sum_{lm} c^g_{l,m} Y_{l,m}(\vec{w}) \right) \left( \sum_{lm} c^h_{l,m} Y_{l,m}(\vec{w}) \right) \, d\vec{w}
\]

\[
= \sum_{l_1,m_1} \sum_{l_2,m_2} \sum_{l_3,m_3} c^f_{l_1,m_1} c^g_{l_2,m_2} c^h_{l_3,m_3} \int_S Y_{l_1,m_1}(\vec{w})Y_{l_2,m_2}(\vec{w})Y_{l_3,m_3}(\vec{w}) \, d\vec{w}
\]

\[
= \sum_{l_1,m_1} \sum_{l_2,m_2} \sum_{l_3,m_3} c^f_{l_1,m_1} c^g_{l_2,m_2} c^h_{l_3,m_3} C_{l_1,m_1,l_2,m_2,l_3,m_3}
\]

(A.1)

The indices \( i, j \) and \( k \) that were formerly used for brevity were again replaced by corresponding \( l, m \) where \( l \in [0, n - 1] \) and \( m \in [-l, l] \) for an SH of rank \( n \). The tripling coefficients \( C_{l_1,m_1,l_2,m_2,l_3,m_3} \) correspond to Clebsch-Gordan coefficients [SN10] and can be written as:

\[
C_{l_1,m_1,l_2,m_2,l_3,m_3} = \sqrt{\frac{(2l_1 + 1)(2l_2 + 1)(2l_3 + 1)}{4\pi}} \begin{pmatrix}
(l_1 & l_2 & l_3) \\
0 & 0 & 0
\end{pmatrix} \begin{pmatrix}
l_1 & l_2 & l_3 \\
m_1 & m_2 & m_3
\end{pmatrix} \]

(A.2)

The general definition of the Wigner 3j symbols is very complicated and therefore omitted here. It can be found in [SN10]. For a given SH rank, the coefficients can be precalculated and the resulting matrix is sparse. For the rank 2 SH used in this thesis, the non-zero Wigner 3j symbols are:
\begin{align*}
\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} &= \frac{1}{\sqrt{\pi}} & \begin{pmatrix} 0 & 1 & 1 \\ 0 & -1 & 1 \end{pmatrix} &= \frac{1}{\sqrt{\pi}} \\
\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} &= \frac{1}{\sqrt{\pi}} & \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix} &= \frac{1}{\sqrt{\pi}} \\
\begin{pmatrix} 1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} &= -\frac{1}{\sqrt{\pi}} & \begin{pmatrix} 1 & 1 & 0 \\ -1 & 1 & 0 \end{pmatrix} &= -\frac{1}{\sqrt{\pi}} \\
\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} &= \frac{1}{\sqrt{\pi}} & \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} &= \frac{1}{\sqrt{\pi}} \\
\begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & -1 \end{pmatrix} &= -\frac{1}{\sqrt{\pi}} & \begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \end{pmatrix} &= -\frac{1}{\sqrt{\pi}}
\end{align*}

Inserting these coefficients into Equation A.1 reduces the sum of $4^3 = 64$ products to a sum of ten products. The equation can be rewritten as:

\[
\int_S \tilde{f}(\vec{w}) \tilde{g}(\vec{w}) \tilde{h}(\vec{w}) \, d\vec{w} = -\frac{1}{2\sqrt{\pi}} \sum_{i=1}^{4} c_i^p c_i^h
\]

where the new SH coefficients $c_i^p$ are calculated from $c_i^f$ and $c_i^g$ as:

\[
\begin{align*}
    c_1^p &= c_2^f c_4^g - c_3^f c_3^g + c_4^f c_2^g - c_1^f c_1^g \\
    c_2^p &= c_1^f c_4^g + c_4^f c_1^g \\
    c_3^p &= -c_1^f c_3^g - c_3^f c_1^g \\
    c_4^p &= c_1^f c_2^g + c_2^f c_1^g
\end{align*}
\]
Bibliography


Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 29.04.2015