Real-Time Global Illumination in Web-Browsers

Marcus Bertilsson[1], Hannes von Essen[1], Daniel Hesslow[1], Niklas Jonsson[1], Simon Moos[1], and Olle Persson[2]

[1]Computer Science and Engineering, Chalmers University of Technology
[2]Computer Science and Engineering, University of Gothenburg

May 2018

Göteborg, Sweden

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF
GOTHENBURG

Real-Time Global Illumination in Web-Browsers

Marcus Bertilsson, Hannes von Essen, Daniel Hesslow, Niklas Jonsson, Simon Moos, and Olle Persson

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2018

Real-Time Global Illumination in Web-Browsers
Marcus Bertilsson, Hannes von Essen, Daniel Hesslow, Niklas Jonsson, Simon Moos, and Olle Persson.
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Global illumination is important for a rendered image to appear realistic. Accurate global illumination can be achieved but most algorithms are very computationally expensive and only suitable for offline rendering where long rendering times can be allowed, and not for real-time rendering where many images every second is needed. We have looked at the limited case of global illumination for certain real-time applications in web-browsers. We present a comparative study of three academic papers that describe algorithms that might be suitable and successfully implemented and adapted them to our context. The results vary in terms of real-time performance, precomputation time and visual fidelity. One algorithm produced results that are not very accurate compared to a reference image but might be visually convincing enough while having the advantage of very high performance and no precomputation. The other two algorithms both produce results of higher fidelity but are generally slower in real-time and require some precomputation. One algorithm requires a couple of seconds at most and the other minutes or even hours depending on the scene. Finally, we conclude that all three techniques are suitable for rendering real-time global illumination, but they have different pros and cons.

Keywords: global illumination, real-time rendering, WebGL, computer graphics

Real-Time Global Illumination in Web-Browsers
Marcus Bertilsson, Hannes von Essen, Daniel Hesslow, Niklas Jonsson, Simon Moos, and Olle Persson.
Department of Computer Science and Engineering
Chalmers University of Technology

Sammanfattning

Global illumination är viktigt för att en renderad bild ska se realistisk ut. Verklighetstrogen global illumination är möjligt men de flesta algoritmer är väldigt beräkningsmässigt tunga och passar endast för offlinerendering där långa renderingstider kan tillåtas, och inte för realtidsrendering där flera bilder per sekund behövs. Vi har tittat på det begränsade fallet av global illumination i vissa typer av realtidsapplikationer i webbläsaren. Vi presenterar en jämförande studie av tre vetenskapliga publikationer som beskriver algoritmer som kan vara passande, och implementerade dem i vårt kontext. Resultaten varierar i realtidsprestanda, förberäkningstid, och visuell exakthet. En av algoritmerna producerar resultat som inte är särskilt visuellt exakta jämfört med en referensbild, men som och har fördelarna av hög prestanda och avsaknaden av förberäkningar, och resultatet kan vara övertygande nog. De andra två algoritmerna producerar båda resultat av högre exakthet men är oftast långsammare i realtid och kräven viss förberäkning. Den ena algoritmen kräver endast ett par sekunder och den andra minuter eller till och med timmar, beroende på vilken scen som används. Vi drar slutsatsen att alla tre tekniker passar väl för att rendera global illumination i realtid, men de har alla olika styrkor och svagheter.

List of Abbreviations

GI	Global illumination
FPS	Frames per second
BRDF	Bidirectional reflectance distribution function
SH	Spherical harmonics
MRT	Multiple render targets
GPU	Graphics processing unit
RSM	Reflective shadow map
SVD	Singular-value decomposition
tSVD	Truncated singular-value decomposition
GV	Geometry volume
LPV	Light propagation volume
SPP	Samples per pixel

Contents

1 Introduction

Global Illumination (**GI**) algorithms in computer graphics are techniques for rendering 3D images that take into account not only the direct illumination from light sources but also the *indirect* illumination caused by light that is reflected from the directly lit objects onto nearby surfaces. For example, an object in a room lit by a single light source will cast a shadow where it occludes the direct light from hitting the floor. If only direct light is taken into account, this shadow will be pitch black because there is no light reaching it. If indirect light is included, the light that hits the wall behind the object will bounce into the shadow region, giving it a brighter, more nuanced appearance. Indirect light includes not only this first bounce but also secondary, tertiary, and potentially infinitely many light bounces, and its effects are not limited to brightening shadows but also include such phenomena as color bleeding–when objects are colored by the bounced light from nearby colored surfaces. Global Illumination can have a substantial effect on the perceived realism of the rendered images, but most algorithms that produce realistic results have been highly computationally intensive and not suitable for real-time rendering where many frames per second have to be rendered for the motion to appear fluid. The application of these algorithms has instead traditionally been limited to *offline* use cases, such as photo-realistic images and 3D animated movies, where each frame is allowed to take several hours or even days to render.

In recent years however, the introduction of novel algorithms as well as the continuing growth of computational power is beginning to change this, making Global Illumination feasible for real-time applications. There are a variety of solutions that seek to approximate the effects of GI, each with different levels of fidelity and with different limitations. We explore techniques for global illumination in the specific use case of *largely static scenes with some dynamic geometry*, in a web-browser setting. This use case may be found in for example interior design software, where the basic structure of a room may be predetermined (static) and the furniture interactively positioned and customized by the user (dynamic). The deployment as a web application is an increasingly popular choice and has many important advantages: it inherently provides a strong cross-platform support and facilitates quick access and dissemination to a large audience. At the same time, it does bring with it some technical limitations, and many of the algorithms have been tested only in desktop environments, which makes real-time GI development for the web-browser an interesting area in need of further investigation.

1.1 Aim

The aim of this project is to investigate and compare possible real-time global illumination techniques, with the goal of finding the optimal technique for our use case. The use case is more precisely defined in terms of a set of criteria for how techniques should be able to perform. An optimal technique should...

1. be able to produce reasonably realistic images with global illumination that are comparable in visual fidelity to a ground-truth high-end ray tracer
2. allow for objects and light sources in the scene to be moved
3. be fast enough to be used in interactive, real-time graphics
4. be implementable in a web-browser

1.2 Scope

Optimally all previously existing techniques that fulfill the criteria (see section 1.1) should be evaluated for us to gain the best possible overview. However, given the limited amount of resources available we have no possibility of doing that. For this reason we have had to limit the number of techniques evaluated to three promising alternatives. The choices made are of course critical to our results so the decision making process was thorough and is documented in section 3.

The fact that the technique should be implementable in a web-browser (criterion 4, section 1.1) limits the choice of API to either WebGL 1.0 or WebGL 2.0. Although not fully supported in all web-browsers [1], WebGL 2.0 provides some features that we consider instrumental to achieving the goal, such as Multiple Render Targets (see section 2.7). We therefore only require of the techniques that they are implementable in WebGL 2.0.

1.3 Methodology

The main focus of this project was to implement and evaluate different algorithms for real-time global illumination in web-browsers. A number of algorithms were investigated and three were chosen to be implemented. The implementations are described in detail in their respective sections and will, in line with the evaluation criteria, be compared in terms of

- Visual fidelity compared to a reference image

- Handling of dynamic lights and objects
- Average frame rendering time in milliseconds
- Precomputation time

As the project was targeted towards 3D-rendering in web applications, the OpenGL-based WebGL 2.0 [2] was a natural choice and also suitable to the group as it is implemented using JavaScript which the group members had previous experience with. PicoGL (v0.8.8) [3] was used to speed up the WebGL development by abstracting away some of the boilerplate code. PicoGL is a library that adds an additional layer to the WebGL API, providing easier access to the functionality of WebGL while still allowing full access to lower-level constructs.

2 Relevant Theory

2.1 Radiometry in Computer Graphics

Radiometry is the study of the measurement of light [4]. In radiometry, light is often though of as the received/emitted energy per unit of time, i.e. flux or radiant power Φ (Watt). Because light can be colored, the representation is often divided into the red, green and blue components so that the radiometric quantities are stored as RGB vectors (in contrast to usual RGB pixel values, radiometric RGB values can take values above 1.0 – representing light that is brighter than a computer screen) [5]. In addition to power, two important radiometric quantities in computer graphics are radiance and irradiance.

Radiance, denoted L, measures power per area per solid angle. For an infinitesimal area and infinitesimal solid angle, radiance effectively measures the power Φ traveling along a single ray of light. A ray is parameterized as $r(t) = \mathbf{o} + t\mathbf{v}$, with origin $\mathbf{o}$ and normalized direction $\mathbf{v}$ for a distance of t from the origin, and is often used to simulate light. The color of a given pixel on the screen directly corresponds to the incoming radiance at the camera's origin in the direction corresponding to the pixel.

Irradiance, denoted E, measures flux per area incident on a surface [5, ch. 5]. This means that light rays that are not perpendicular to the surface have to be multiplied by the cosine of the angle between the surface normal and the light ray in order to obtain the component of the light that is perpendicular to the surface. While radiance can be seen as measuring the incoming light of a single ray, irradiance measures the combined light of the incoming rays from all directions around the surface. This quantity is important because the defining characteristic of Lambertian diffuse surfaces (the most common type of material in computer graphics) is that the outgoing radiance is proportional to the incoming irradiance on the surface. The outgoing radiance from such a surface is thus the same in every direction, and is given by Equation 1 [5, ch. 5]

$$\frac{E \otimes \mathbf{a}}{\pi} \tag{1}$$

where $\otimes$ represents element-wise multiplication and $\mathbf{a}$ is the surface's *albedo*. Albedo is the proportion of received light that is reflected rather than absorbed by the material, and is what we would normally call the object's color. For instance, an object with albedo $rgb = (0.0, 1.0, 0.0)$ would reflect all of its incoming green light, and absorb all its red and blue light, giving it a green appearance.

2.2 The Rendering Equation

In 1986 Kajiya presented the Rendering Equation as a generalized version of many previously known rendering algorithms [6]. By solving the equation it is possible to retrieve a physically correct render of most materials.

There exist many variations of the Rendering Equation. In this paper the following definition is used (Equation 2):

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_o, \omega_i) L_i(x, \omega_i) \cos(\theta_i)\ d\omega_i \tag{2}$$

using the following symbols:

- $L_o(x, \omega_o)$ Total outgoing radiance from point x in direction ω_o
- $L_e(x, \omega_o)$ Outgoing emitted radiance from point x in direction ω_o
- $\int_\Omega d\omega_i$ Integral over all incoming directions ω_i in the surface normal oriented hemisphere
- $f_r(x, \omega_o, \omega_i)$ The bidirectional reflectance distribution function (**BRDF**) for the material, evaluated at point x. The BRDF describes the proportion of incoming light from ω_i that leaves in the direction ω_o.
- $L_i(x, \omega_i)$ Incoming radiance at point x from direction ω_i
- $\cos(\theta_i)$ The cosine of the angle between the surface normal and ω_i, modeling light attenuation, i.e. the fact that the light becomes more spread out (and therefore less intense) as the angle to the surface normal increases.

Note that the the L_i term is evaluated through the Rendering Equation as well, i.e. the equation is infinitely recursive, and has no closed form. It does, however, converge and can therefore be approximated though numerical evaluation [6].

The BRDF defines the material of a surface. A common BRDF for diffuse surfaces is the Lambertian BRDF, defined as $f_r(x, \omega_o, \omega_i) = \frac{1}{\pi}$ for a white material, or $f_r(x, \omega_o, \omega_i) = \frac{\mathbf{a}}{\pi}$ for a material with albedo $\mathbf{a}$.

2.3 Real-Time Rendering

Real-time rendering usually refers to three-dimensional (**3D**) computer graphics rendering which is performed in real-time [5, ch. 1].

An important aspect of real-time rendering is interactivity: ”An image appears

on the screen, the viewer acts or reacts, and this feedback affects what is generated next." [5, p. 1]. For this to work, new images have to be generated at a high frame-rate. [5] argues that 15 frames (i.e. images) per seconds (**FPS**) is considered real-time, however it is also mentioned that higher rates are preferred since it decreases the response time of the user. Studies such as [7] and [8] similarly indicate that that the viewer's ability to react is improved with frame rate, and argue for 30-60 FPS as a suitable frame rate for interactivity, which translates to approximately 16.7 to 33 milliseconds of time to generate each image.

Note that the definition of real-time rendering does *not* preclude precomputation steps; for instance computations that are specific to a certain static scene can be performed beforehand and saved together with the scene in order to speed up the real-time execution of the program. The only piece of data that has to be generated in real-time is the image shown on the screen.

2.4 Global Illumination

Global Illumination (**GI**) refers to a collection of techniques for adding indirect light to surfaces in an image [5]. An image without GI has only direct light, i.e. light that reaches surfaces directly from light sources. Indirect light, on the contrary, is light that hits surfaces after bouncing one or more times on other surfaces. The difference can be explained in terms of the Rendering Equation (Equation 2) as the recursion depth for the L_i term. While GI requires full evaluation of the equation (or some approximation of it), images without GI only require that L_i is evaluated once for each direction ω_i, i.e. light incoming directly from light sources.

The effect of GI can be observed in Figure 1; note especially the surfaces that are not directly visible from the light source, and how the color of surfaces are affected by the color of neighbouring surfaces.

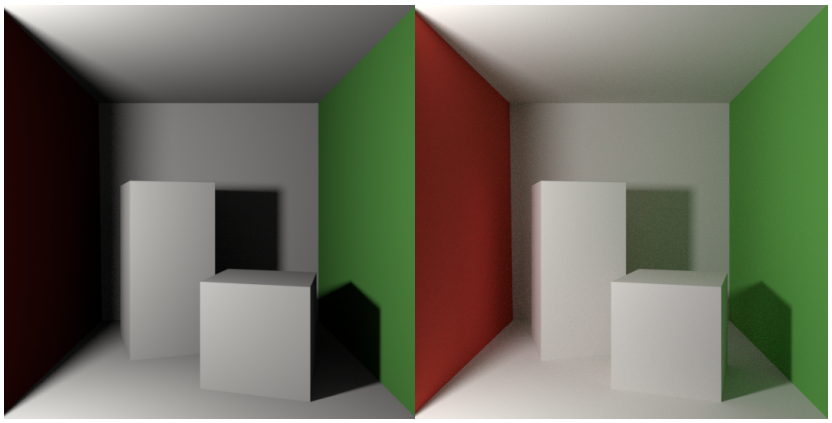

Figure 1: The Cornell Box scene [9] rendered twice in Blender Cycles Renderer with all settings except global illumination (i.e. number of light bounces) kept the same. Left: without global illumination. Right: with global illumination.

While in the real world there exists no distinction between direct and indirect light, the distinction is important in computer graphics since there is a significant difference in the amount of computations needed for the two types of light. In offline rendering the high computational requirement for indirect light is not a significant problem because it can be correctly simulated with good results if enough time is spent. In the context of real-time rendering on the other hand, GI is difficult since a frame should not require more than 33 milliseconds to render (see section 2.3).

Since the Rendering Equation is infinitely recursive and has no closed form (see section 2.2) approximations are imperative for achieving real-time performance, and real-time GI solutions often mostly differ in what approximations they use and how they are applied. One common approximation is based on the precomputation of certain aspects of GI offline, to simplify the calculations performed in real-time.

2.5 Light Fields

We have previously discussed how radiance can be used to describe the light flow of a single ray of light at a point in space (see section 2.1). Viewed as a function of position and direction, radiance can be used to describe the light flow at every point and in every direction in a scene. This function is called the light field of the scene [10].

2.6 Spherical Harmonics

There are many possible ways to represent the light field around a point in space. A common approach is to use a mapping from the three-dimensional vectors on the unit sphere to two-dimensional points on a rectangle (e.g. cube maps or octahedral maps), allowing the light field at a point to be stored as an image texture [5]. An alternative method is to project the surroundings onto the spherical harmonics (**SH**) basis functions. This yields a compact representation in the form of a few numbers – the coefficients of the included basis functions.

A spherical function is a function that takes as input a three-dimensional direction on the sphere. This direction can be specified by two angles or equivalently by a three-dimensional unit direction vector. Spherical harmonics is a series of specific spherical functions that together constitute an orthogonal basis over the sphere, which means that an arbitrary spherical function can be represented as a linear combination of them [11]. The function can be approximated arbitrarily well by choosing to include a large enough number of spherical harmonics basis functions. On the other hand, including only a small number of basis functions (such as the 16 in Figure 2) yields a low frequency approximation of the function, while only requiring the coefficients of this small number of basis functions to be stored.

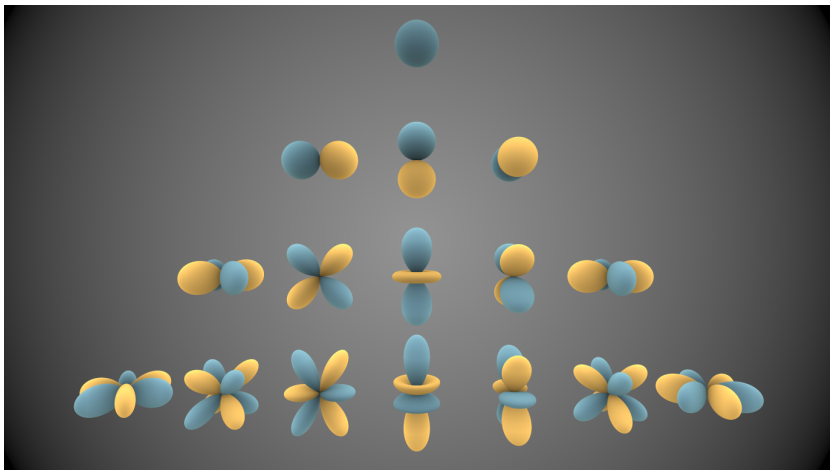

Figure 2: Visualization of the first 16 spherical harmonics basis functions. Each function is represented here by a color-coded surface around its center. For a given direction, the magnitude of the function is given by the distance from the center to the surface in that direction, with blue indicating positive values and yellow indicating negative values. For example, the first function has a constant positive value in every direction and the three functions in the second band are positive in one direction and negative in the other, varying smoothly in-between. Image generated using [12].

Each SH basis function is characterized by its band index (l) (the horizontal rows in Figure 2, starting at band zero) and the basis function index within that band (m) – with zero at the center column in the figure, decreasing to the left (-1, -2, ...) and increasing to the right (1, 2, ...) [11]. Each band corresponds to polynomials of that degree. For our purposes, it is convenient simply to refer to the basis functions with a single index, going from left to right in each row of the figure starting at the top; i.e. assigning the index $i = l(l + 1) + m$. We will denote the ith SH basis function $Y_i(d)$, where d is the input direction. Assume that we have a spherical function $f(d)$, and denote its corresponding SH coefficients c_i, $i = 0, 1, 2, ...n - 1$. The approximated value of f in the direction d is then evaluated by the following linear combination:

$$f(d) \approx \sum_{i=0}^{n-1} c_i \cdot Y_i(d) \tag{3}$$

2.6.1 Projecting onto the Spherical Harmonics Basis

In order to utilize the compact representation of spherical harmonics, we need to be able to convert an arbitrary spherical function (such as the incoming radiance around a point as a function of the direction) into the coefficients of the SH basis functions that can be used to approximate the function, i.e. we need to project the function into the SH basis.

A vector v can be projected onto another vector u using the dot product, as described by the *projection formula*: $\frac{v \cdot u}{|u|}$. For functions, the dot product is not defined. Instead, to project a function onto another function we must use the generalized version – the inner product, denoted $\langle f, g \rangle$ where f and g are the operands. To project f onto g the formula then becomes $\frac{\langle f,g \rangle}{|g|}$. For functions, the inner product is defined as $\langle f, g \rangle = \int_x f(x)g(x)$. This integral can be approximated with numerical integration. We are interested in projecting an arbitrary function f onto the spherical harmonics basis. Since the SH basis functions are orthogonal it is possible to approximate each of these functions separately; the ith SH coefficient is given by the projection of f onto the ith SH basis function $Y_i(d)$. Additionally, SH basis functions are normalized which means that the division by the norm is not necessary. If we sample uniformly each sample has the same weight and the formula becomes

$$c_i = \int_{d \in \Omega} Y_i(d) f(d) \approx \sum_{k, d_k \in \Omega}^{n} \frac{4\pi}{n} Y_i(d) f(d) \tag{4}$$

Here c_i denotes the coefficient for the ith basis function in the approximation,

d a direction on the sphere Ω, d_k the kth uniformly sampled direction, $Y_i(d)$ ith basis function evaluated in the direction d and $f(d)$ the spherical function evaluated in the direction of d.

However, to use rasterization provided by graphics hardware this formulation is not quite sufficient. It is not possible to uniformly sample the function since no linear functions that maps the plane onto the sphere exists. Instead a common representation of spherical functions in computer graphics is cube maps. However, when using cube maps the sampling is no longer uniform (pixels near the center of the cube faces map to larger areas on the sphere compared to pixels near the corners). Therefore we must weight the samples by their projected area on the sphere, as is done in [11].

2.6.2 Ringing

Storing a function as SH coefficients has many advantages over traditional texture-based methods: the approximation is generally very smoothly varying, with none of the pixelated artifacts seen when using low-resolution textures, and it is possible to perform seamless rotations through mathematical operations on the coefficients. However, there are also unique problems associated with spherical harmonics representation, one of the most common being *ringing* artifacts [11]. This is often seen when there are rapid and large variations in the spherical function to be approximated and too few SH basis functions are used, and is caused by the included SH basis functions' limited ability to represent these sharp variations. Because the basis functions generally have both positive and negative contributions (see Figure 2), the need to approximate an extreme positive value in one direction may inadvertently cause a large negative value in the other direction. There have been attempts at mitigating ringing artifacts through various techniques such as *windowing* [11].

2.7 Multiple Render Targets

A render target texture is a modern alternative to standard rendering where all rendering is done using a default frame buffer. A frame buffer is a place in memory containing data that is to be rendered to the screen. A render target texture is a user defined frame buffer object which makes it possible to render objects independently of the rest of the scene. A benefit of using a render target texture instead of the default frame buffer is that effects can be applied to the rendered image as the target texture can be passed into a shader where the data can be manipulated [5].

Multiple render targets (**MRT**) is an extension to render target textures in modern computer graphics which allows the fragment shader to output multiple

vectors, each to its own render target, which can improve performance; instead of rendering to one target texture followed by a render to another target texture it is possible to render to multiple target textures at once, removing the need to recompute time-consuming vertex transformations that are shared between textures [5].

For OpenGL and WebGL the extension `ARB_draw_buffers` [13] allows for rendering to multiple render targets and is available to use in all WebGL 2.0 contexts [2, ch. 3.7.11].

2.8 Shadow Maps

Shadow mapping [14] is an algorithm used to generate shadows. The algorithm makes use of a depth map where at every pixel the distance between the corresponding world space position and the light source is stored. To test whether a pixel is in shadow or not we render a depth map from the light's point of view followed by a render of the scene from the camera's point of view. The pixel's position in the camera's point of view is then transformed into light space and if the depth is greater than the value stored in the depth map the pixel is in shadow and if the depth is less than the value stored in the depth map the pixel is lit [5]. A shadow map can be visualized by rendering the depth values on a scale from black (near) to white (far) as seen in Figure 3.

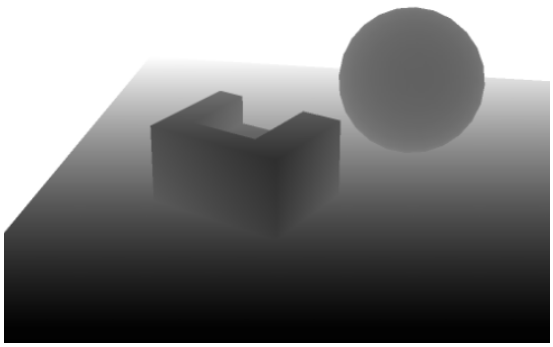

Figure 3: Image of a depth map. The pixel color indicates the distance to the rendered point in the scene from the perspective of the light source [5].

2.9 Reflective Shadow Maps

The reflective shadow map technique was first described in 2005 [15] by Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps (**RSM**) is an extension to regular shadow maps where aside from the depth, the world space coordinates, the world space normals and flux (the radiant power of a light source) are also stored in different textures, together composing the reflective shadow map.

The idea behind RSMs is that we consider each pixel of a shadow map as an indirect light source that generates a single bounce of indirect illumination. This means that if we have a single point light source all the indirect illumination generated by a single bounce will be visible in its shadow map. By utilizing this method we are able to efficiently and in parallel sample secondary light sources on a modern graphics processing unit (**GPU**).

The world space coordinates and the normals can be fetched from the geometry as usual. The flux has to be calculated depending on the type of light source. The reflected flux for directional lights is calculated by multiplying the surface color with the color of the light source and the clamped dot product between the light direction and the surface normal, as shown in Equation 5.

$$\phi_p = c_l \cdot d_p \cdot (n_p \dot{+} -v_l) \tag{5}$$

Here ϕ_p is the resulting flux of a point p, c_l is the color of a light source l, n_p is the normal of a surface point, v_l is the directional vector of a light source and $\dot{+}$ denotes the dot product clamped between 0 and 1. When rendering from the perspective of a directional light source, an orthographic projection matrix is used. Flux could easily be calculated for other types of light sources by also accounting for the perspective of the light source by instead using a perspective projection matrix.

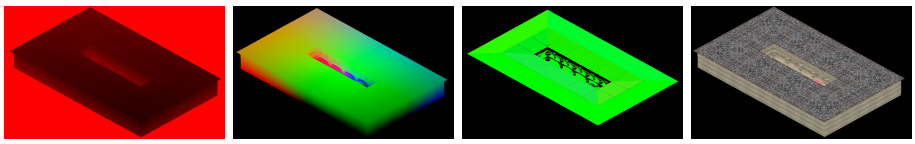

Figure 4: From left: the depth, world space positions, normals, and flux of the RSM for an example scene.

The different textures of the RSM for an example scene is shown in Figure 4. From these it is trivial to fetch the desired values from the different textures.

2.10 Variance Shadow Maps

Variance Shadow Maps [16] is another extension of regular shadow maps that differ in that they can be filtered just like color textures. Filtering (such as blurring) regular shadow maps does not result in filtered shadows, but it does for variance shadow maps. While regular shadow maps use a boolean test to evaluate if a point is in shadow, variance shadow maps use a statistical test based on the one-tailed version of Chebychev's inequality (Equation 6).

Let X be a random variable with the distribution of the depth values in the filtered shadow maps at the current pixel with mean μ and variance σ^2. t indicates the actual distance from the shading point to the light source which is what should be tested. Given that $t > \mu$:

$$P(X \geq t) \leq \frac{\sigma^2}{\sigma^2 + (t - \mu)^2} \tag{6}$$

If $t < \mu$ the point is not in shadow. Otherwise $P(X \geq t)$ indicates the probability that the point is in shadow. Since this is a continuous and smoothly varying value between 0 and 1 it can be used to indicate how much of the point is in shadow, from not in shadow to fully shadowed.

The mean μ in Equation 6 is the filtered pixel value, i.e. $E(x)$. The variance σ^2 can be calculated as $\sigma^2 = E(x^2) - E(x)^2$. To get $E(x^2)$ the squared depth x^2 is stored in the variance shadow map in addition to the depth x in a separate channel which is filtered identically. Note that while Equation 6 is an *in*equality, it can be an equality in a more constrained case which is similar to the case of variance shadow maps [16, ch. 3.1]. Therefore the inequality can be though of as an equality in the context of variance shadow maps and can be evaluated as such.

3 Choice of Techniques

Given the aim of this project and the constraints set up, it is clear that a major challenge lies in the decision of which algorithms to focus on. Real-time GI is an active research area with many different directions (see [17]), choosing only three to implement and evaluate is not an easy task. Furthermore, the quality of this work is directly dependent on the quality of our choices, since the final results will inevitably be based on what we have implemented. Because of the small sample size out of all existing techniques the choice will undoubtedly be biased. To try to select a set of promising methods we developed a number of heuristics to evaluate possible candidates:

1. Because of the aforementioned breadth of the research field it is important that the three techniques reflect that property in terms of:
   (a) the amount of precomputations needed
   (b) how large changes in terms of dynamic objects and light sources the technique can tolerate
2. All of the three techniques should be relevant today

With these heuristics and constraints the following three techniques were chosen:

1. Real-Time Global Illumination using Precomputed Light Field Probes [18]
2. Real-time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes [19]
3. Light Propagation Volumes (Crysis) [20]

Both [18] and [19] were published in 2017 and are therefore highly relevant. While [20] was published in 2009 it is arguably still relevant, since it requires no precomputing and is in use in popular and current game engines, such as Unreal Engine 4 [21].

Regarding the first heuristic, all of the three chosen algorithms differ significantly. As mentioned, [20] requires no precomputation and related to that supports fully dynamic objects and light sources. On the contrary, [19] requires minutes to hours of precomputation, depending on the complexity of the scene, and for that reason has to be performed in a separate step from the real-time part. This technique fully supports dynamic light sources, but it does not support dynamic objects by itself. However, after reading the paper it became clear that at least some limited support for dynamic objects should be possible, and given the very promising results for static scenes presented by the authors, it was deemed worthwhile to investigate the possibility of adapting the technique

to our semi-dynamic use case. [18] does require precomputation but the time required is somewhere between a few seconds to a minute. It was therefore judged plausible that the precomputation could be performed at load time and therafter partially per frame, hopefully at a rate fast enough for our real-time use case. The technique by itself does *not* support any dynamic objects or light sources at all, but by recalculating the precomputations (partially over multiple frames) in the event of changes, the possibility of achieving the effect of dynamic light sources and objects seemed high enough for the technique to be worth further investigation.

The techniques thus each have their own unique strengths related to our aim, and through evaluating these three techniques we believe we have covered a wide range of the research field with respect to the heuristics set up.

4 Technique 1: Real-Time Global Illumination using Precomputed Light Field Probes

This GI technique was first published in 2017 by McGuire et al. [18], and builds upon the idea of light fields (see section 2.5). By encoding the light field at discrete locations, i.e. light field probes, in a grid, we approximate the continuous light field for the enclosing volume of surfaces that should be shaded. Each probe maps its surroundings in terms of the incident radiance (i.e. the light field), the radial distance to the point closest to the probe in the outgoing direction and the surface normal of that point.

Using the light field and the additional information it is possible to perform ray-tracing in image-space of the probes, but to make the ray-tracing easy and efficient a set of optimizations are applied to the data.

4.1 Octahedral mapping

Octahedral mapping [22] is a way of encoding spherical data into a flat 2D representation. In the case of this technique it is used to map all six sides of a cubemap to a single 2D texture. This is often advantageous to do since it compresses the data into a single texture with one common 2D coordinate system. An illustration of what the mapping performs can be seen in Figure 5, and pseudo code for encoding and decoding can be found in the Algorithms 1 and 2, respectively.

Input: Unit vector v
Output: Octahedral encoded coordinate o on the $[-1, +1]$ square

$\text{norm} \leftarrow |v.x| + |v.y| + |v.z|$
$o \leftarrow v.xy \ / \ \text{norm}$
if $o.z < 0.0$ **then**
  $o \leftarrow (1.0 - |o.yx|) \cdot \texttt{signNotZero}(o.xy)$
end

Algorithm 1: Encoding to the octahedral projection. `signNotZero` returns $+1$ for inputs $x \geq 0$ and -1 otherwise.

There exist other mappings that can achieve similar results, such as the common spherical mapping [22]. A unique feature about octahedral mapping which makes it suitable for use in this technique is that it preserves straight lines. A straight line (e.g. a ray) maps to 1-4 line segments (on separate octahedral faces) with C^0 continuity, also known as a polyline, which is trivial to trace.

Input: Octahedral encoded coordinate o on the $[-1, +1]$ square
Output: Unit vector v

$v \leftarrow \texttt{vec3}(o.x, o.y, 1.0 - |o.x| - |o.y|)$
if $v.z < 0.0$ **then**
  $v.xy \leftarrow (1.0 - |v.yx|) \cdot \texttt{signNotZero}(v.xy)$
end
$v \leftarrow \texttt{normalize}(v)$

Algorithm 2: Decoding from the octahedral projection. `signNotZero` returns $+1$ for inputs $x \geq 0$ and -1 otherwise.

4.2 Probe Image-Space Ray-Tracing

Rays can be traced within a probe by marching the ray along its polyline in octahedral-space and comparing its distance from the probe to the precomputed distance. Ray-marching is a technique where ray-tracing is approximated by iteratively stepping along a line in discrete steps, such as texel-coordinates in image-space. By doing this it is possible to approximate the Rendering Equation (Equation 2) in a way that only depends on the resolution of the precomputed distance and not the number of triangles in the scene, which is a major problem with ray-tracing performance. For the rest of this report this process is referred to as probe image-space ray-tracing.

For the reason discussed above it is possible to use several different resolutions of the precomputed distance. By storing both a high resolution map (such as 1024^2) and a very low resolution map (such as 64^2) very good ray-marching performance can be achieved.

The lower resolution distance map can be used until the ray is close to a surface, where it is swapped with the high resolution map to increase precision. The result of a trace will be HIT if the ray hits a surface, MISS if the ray does not hit any surface, or UNRESOLVABLE if the ray travels behind some surface to a part of the scene which is occluded from the probe. If the result is UNRESOLVABLE, a new probe is selected and the routine is repeated from another probe.

A new probe is selected by considering the probes that create a bounding box around the area in which the previous probe trace terminated. If this is the same bounding box as the last attempt, the probe of the bounding eight probes which is furthest away form the previous probe is selected in an attempt to avoid the occluding geometry. If the previous probe trace terminated in another cube of bounding probes, the same procedure is repeated considering the new probes.

In the case of a HIT, the precomputed surface normal is used to get the angle of the bounced ray. The normal is also used to perform backface testing for the rays.

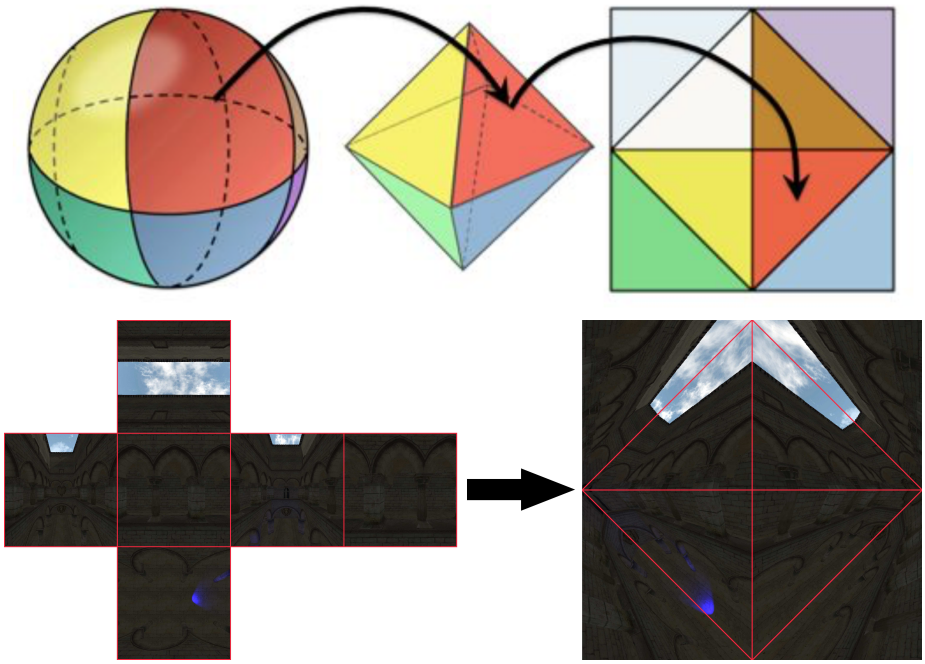

Figure 5: Illustrations of the octahedral mapping. Top [22]: visualization of how sections of the sphere surface maps to triangles in the octahedral representation. Bottom: faces of a radiance cubemap in an example scene [9] mapped to an octahedral represenation. Note that edges (red) do *not* line up between representations!

4.3 Diffuse Indirect Light

By tracing rays across the probes, glossy reflections in world space can be calculated efficiently enough for real-time applications. For diffuse reflections, three algorithms are presented in the paper. The first one is a brute-force Lambertian importance sampling technique where multiple glossy rays are computed to achieve good, but very expensive, results. The second algorithm is an extension and optimization on the first that does not require as many rays to be computed. Given the rendering times specified in the paper [18, ch. 6] we consider both of these algorithms to be too slow for real-time use, though.

The third algorithm–Irradiance with (Pre-)Filtered Visibility–is the main focus of the paper and also the one that is fastest and most suitable for real-time applications with dynamic objects, so this is the algorithm we implemented. For this method, two additional octahedral maps have to be precomputed for each

probe: one Lambertian-prefiltered irradiance map, and one variance shadow map (see section 2.10). Both are generated from the existing radiance and distance cubemaps.

To estimate the incident irradiance at a shading point, the irradiance maps of the bounding box of eight surrounding probes are trilinearly interpolated, combined with two visibility test. Using the variance shadow maps for the probes a smooth visibility check is performed where the probe does not affect the overall irradiance if the shading point is not visible from the probe. Variance shadow maps are used (instead of regular shadow maps) to allow the replication of the "soft"/smooth nature of indirect diffuse light and their shadows. A basic smooth backface test is also performed, as described in Equation 7, to make sure only forward facing surfaces are affected by the indirect light from a given probe.

$$\text{backface-test factor} = \max(0, \hat{\mathbf{n}}_{\text{surf}} \cdot \frac{\mathbf{p}_{\text{probe}} - \mathbf{p}_{\text{surf}}}{||\mathbf{p}_{\text{probe}} - \mathbf{p}_{\text{surf}}||}) \tag{7}$$

where $\hat{\mathbf{n}}_{\text{surf}}$ is the surface normal at the shading point, $\mathbf{p}_{\text{surf}}$ is the position of the shading point, and $\mathbf{p}_{\text{probe}}$ is the position of the probe.

4.4 Implementation Details

The precomputation is performed in three steps for each probe. The first step is to render cubemaps for radiance, surface normals, and distance at the probe location. This is done in one pass using MRT (see section 2.7). The cubemaps are then mapped to octahedrals for the radiance, distance, and normal maps, also in one pass. For the irradiance and filtered distance maps the filtering is performed in the coordinate space of the cubemaps, for convenience, and is mapped directly into their respective octahedral maps.

For filtering the irradiance and filtered distance maps for diffuse indirect light the technique described in Algorithm 3 is used. The uniform distribution of sample points on the sphere is generated on the CPU and accessed in shaders through a uniform buffer. The function `uniformPointOnSphere` in the algorithm simply picks the ith sample in the uniform buffer.

As discussed in section 3, this technique does not by itself allow for dynamic objects and light sources; to make it work we perform the precomputation in the background as changes in the scene are made. While the precomputation needed for a single probe does not take a significant amount of time to perform, it becomes substantial when all probes are considered. To allow the precomputation to be performed dynamically while still retaining real-time performance only a few probes are precomputed per frame over a span of multiple frames.

```
for pixel, uv in pixels to filter do
    n ← octahedralDecode(uv)
    pixel ← vec3(0)
    for i .. number_of_samples do
        d ← normalize(n + s · uniformPointOnSphere(i))
        pixel ← pixel + texture(cubemap, d)
    end
    pixel ← pixel / number_of_samples
end
```

Algorithm 3: The filtering algorithm used for calculating irradiance and filtered distance octahedral maps. s defines the size of the filter kernel; for Labertian irradiance filterering $s = 1$, while for the filtered distance should be kept small, e.g. $s \leq 0.5$.

Turnaround time, i.e. the time for the GI to respect the new world state, now becomes relevant instead, and should be kept as low as possible to achieve good results when visually significant changes occur.

The real-time step is performed using the supplemental shader code provided with the paper. However, changes had to be done to make it compile in GLSL version `300 es`, which is the most recent version supported in WebGL 2.0 [2, ch. 4.3].

5 Technique 2: Real-time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes

The second technique, introduced by Silvennoinen and Lehtinen in 2017 [19], is also probe-based. It assumes, in its original formulation, that the scene geometry is static which allows for a large portion of the algorithm to be precomputed. By using a sophisticated spatial interpolation of the probes that is more precise than previous versions, fewer probes are needed to get a good result.

To render a scene all we need to know is the radiance at the scene's surfaces; the light field in its entirety, which covers every point in space, is not necessarily needed. Because the scene surface is assumed to be static, we can define a set of static points on the scene surface beforehand, what will be referred to as the receivers. Assuming that the scene surface is Lambertian diffuse, the radiance from a receiver is the same in every direction (see section 2.1), and so can be represented by a single RGB value. The scene's lighting could then be represented as a vector of each receiver's radiance value. Using this representation, the global illumination problem could be described as a matrix M where each element M_{ij} is a factor describing how much a point light at the position of receiver i increases the light at the position of receiver j, i.e. how the radiance at j is affected by the radiance at i. This matrix would describe the relationship between all pairs of surface points in the scene, and would allow a global illumination simulation to be run by starting with a vector of radiance values assigned to the receivers that represents the previously known direct light situation, and then iteratively propagating the light between all receiver pairs, by multiplying our irradiance vector with the matrix M from the left. Similarly, the vector representing n+1 bounces would be obtained by left-multiplying the vector representing n bounces. However, to get a good result the number of receivers would have to be large; in typical scenes within an order of magnitude of 10^5. This would imply that the matrix M would have on the order of 10^{10} entries, and take approximately 40GB to store densely.

For this reason, probes are introduced into the scene. These are a small set of points in free space where the incoming radiance from the receivers is gathered in every direction. We can then define the indirect illumination at a receiver as a function of the probes. This function is described using a matrix and will be referred to as the *local transport matrix*. As noted by Silvennoinen and Lehtinen, this representation should in principle store exactly the same information as the other representation.

It is still not immediately obvious why this problem formulation simplifies the problem, because it would still theoretically require each probe to store the incident radiant intensity in all directions. However, to make the problem practical,

each probe stores only an approximation of its incident light field. Silvennoinen and Lehtinen use the first few spherical harmonics (see section 2.6) as a basis for their approximation, effectively applying a low pass filter over the probe. While the approximation introduces some errors in the solution, it is still a suitable choice because indirect light tends to be more slowly varying than direct light.

5.1 Theory

To find the radiance of the receivers in a certain direction ω in terms of the probes we might be tempted to simply spatially interpolate the radiance in direction ω at nearby probes:

$$L(x, \omega) = \frac{\sum_i w_i(x) L(p_i, \omega)}{\sum_k w_k(x)} \tag{8}$$

where $w_i(x)$ is the spatial interpolation weight for the ith probe, taking the value 1 at the position of the probe and gradually decreasing for positions further away until it reaches 0 for values outside its *support radius* r. More precisely, let

$$f(t) = \begin{cases} 2t^3 - 3t^2 + 1, & \text{if } 0 \leq t \leq 1 \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

Then the spatial interpolation weight is defined as

$$w_i(x) = f(\frac{||x - p_i||_2}{r}) \tag{10}$$

where p_i is the position of the ith probe and r is the support radius mentioned above.

While some early work indeed took the approach of simple spatial interpolation (Equation 8) [23] [24], it has two problems. Firstly, querying the probes using the same direction ω is incorrect because the probes are not at the same position as the receiver and therefore need to look in slightly altered directions in order to see what the receiver sees in that direction. This is corrected by querying the probes not in the direction ω but in the direction of the point h seen by the receiver in the direction ω. This direction is dependent on the probe position and is denoted $\psi_i(\omega)$ where i denotes the index of the probe. The interpolation using this correction (sheared spatial interpolation) is illustrated in Figure 6.

The second problem is that the interpolation is susceptible to *light leakage* – when light incorrectly passes through objects. This is a problem both for the

simple and the sheared version above, and is illustrated for the sheared version in Figure 7 together with the solution – the visibility-aware sheared version. The visibility-aware sheared version incorporates a binary visibility factor $V_i(\omega)$ that causes probes that do not see the point h in the direction $\psi_i(\omega)$ to be left out of the interpolation. The final formula, with both corrections above, becomes:

$$L(x,\omega) = \frac{\sum_i w_i(x) V_i(\omega) L(p_i, \psi_i(\omega))}{\sum_k w_k(x) V_k(\omega)} \tag{11}$$

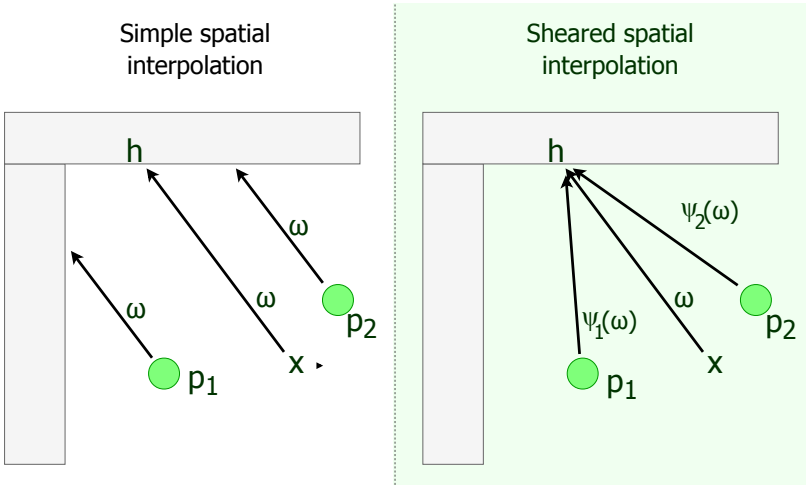

Figure 6: Illustration of simple spatial interpolation and sheared spatial interpolation. We can see in the simple version that the arrows from the probes (p_1 and p_2) in the direction ω hit the surface at points far from the true hit point h for the receiver x in that direction. While this error could be mitigated by increasing the number of probes, the sheared version instead solves the problem by querying the probes in the direction of h, $\psi_i(\omega)$, giving a correct interpolation even for sparse probe sets.

5.2 Computing Receiver Locations

First, the entire scene is mapped to a to a texture so that each 3D triangle in the scene corresponds to a 2D triangle in the texture and so that no two triangles are overlapping in the texture. This is often referred to as *uv unwrapping*, because it can be thought of as folding the surface that wraps around the 3D model into a flat surface, whose 2D coordinates are called u and v rather than x and

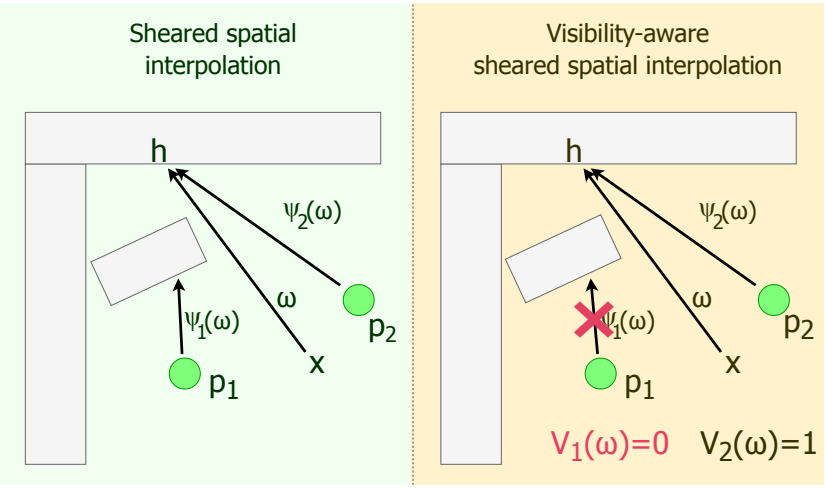

Figure 7: Illustration of sheared spatial interpolation and visibility-aware sheared spatial interpolation. Without visibility awareness, probe p_1 contributes radiance from the block instead of from the desired surface point h. This is incorrect and could lead to light leakage. With visibility awareness the probe p_1 is ignored because it cannot provide information about the radiance from h.

y. By creating this mapping we can conveniently define the receivers as the points on the scene surface (in world space) that correspond to each texel on the map. To get these world space positions we must rasterize the triangles and interpolate the world positions. However, it is not sufficient to only include the texels that overlap with a triangle; we must rather include all texels that will be included when interpolating light for all positions in the triangle. This is equivalent to adding all texels which have a triangle within a manhattan distance of a pixel size from the pixel's center. We approximate this by using conservative rasterization followed by a runtime padding of the lightmap. Since it is difficult to create a good uv unwrapping, the library thekla_atlas [25] was utilized.

5.3 Computing Probe Locations

In order to use the probes as effectively as possible, the probes should be well spread out to provide a good coverage of the scene. We use the approach described by Silvennoinen and Lehtinen [19]. First the scene is *voxelized*. A voxel is the three-dimensional analog of a pixel. While pixels are arranged in a 2D grid, voxels are arranged in a 3D grid, and voxelizing the scene means that, starting with a 3D grid enclosing the entire scene, we fill all voxels that contain part of the scene surface, giving us a blocky approximation of the surface that is

simpler to work with than the surface itself. The probes should be placed outside any object, but the previous voxelization only fills the voxels that intersect with the surfaces in the scene. In order to be able to place probes only on outside of object surfaces, we want both the voxels intersecting with the scene surfaces and voxels in the interior of the scene surfaces to be filled. This is done by flood-filling the scene, which means that groups of neighbouring empty voxels all get filled.

In the next step, we go through all empty voxels that have a filled neighbor. For each of these empty voxels, we generate a *candidate probe* at its center. After this, we will have a set of candidate probes that all (1) lie outside of objects and (2) lie close to the scene surfaces. This set is typically very large, and needs to be reduced to get a suitable number of probes. This is done iteratively, removing the probe from the densest region of probes each time, which leads to an even distribution of probes with an approximately constant density. The density is measured in the following way:

$$\text{density}(p_k) = \sum_i w_i(p_k) \tag{12}$$

where $\text{density}(p_k)$ is the density at probe p_k and w_i is the weight function defined in Equation 10.

5.4 Interpolation of the Probes

As it would be computationally expensive to include all probes in the interpolation of probes for a receiver, it is desirable to select a smaller subset of the probes for each receiver. To decide which probes each receiver should interpolate between Silvennoinen and Lehtinen use the support radius for each probe – the r in Equation 10. A receiver uses a probe in the interpolation only if the receiver lies within the probe's support radius. This means that the number of probes used can vary between receivers. However, at least a few probes should be included in the interpolation in order to avoid discontinuities. Experimentally we have found 6 to be sufficient. To achieve this, [19] use an average of 10 probes to interpolate between.

To reduce the amount of computation, instead of using a constant probe support radius, we chose to use a variable radius, for each receiver. This way we can set each receiver's radius such that each receiver interpolates between exactly the minimum number of probes necessary for the desired quality. If we choose $n-1$ probes to interpolate between, we set the radius of the receiver to the distance to the nth closest probe. This ensures smooth transitions between the probes and considerably reduces the computation needed.

5.5 Computing the Local Transport Matrix

During the computation of the local transport matrix we will need to query if many different points are visible from a given probe. To facilitate this query a cube-map with depth is generated for all probes, which can be considered a point-light shadow map.

The local transport is computed in two passes for each direction we want to sample. First we compute the normalized weights W_i for all probes to be interpolated, modulated by visibility, Equation 13.

$$W_i = \frac{V_i w_i n \cdot (h - p)}{\sum_k V_k w_k} \tag{13}$$

where h is the hit point, p is the world space position of the receiver and n is the normal of receiver.

The spherical function is then approximated with spherical harmonics by multiplying it with the basis functions in the reprojected, $h - p$, direction. While it is possible to compute this with a ray-tracer, we choose to use traditional rasterization to utilize the parallelism of the GPU. To achieve this the spherical harmonics where projected from cube maps as described in 2.6.1.

5.6 Gathering Radiance at the Probes

To be able to effectively query the radiance at surfaces in the scene, a lightmap is generated. Each probe stores its incident light field as a set of spherical harmonics coefficients. To reach this compact representation we need to perform numerical integration as described in 2.6.1. Thus we need to sample the incident radiance in a set of random directions. For each of these random directions, the incident radiance is found by looking at the closest intersection point with the scene in this direction – the color of the scene surface at this point is the desired radiance. Performing this ray-tracing in real-time would be prohibitively expensive. Instead, by deciding on a fixed set of directions, the lightmap uv coordinates corresponding to the hit point in each of these fixed directions can be precomputed beforehand. Then at run-time, the expensive ray-tracing operation for each sample is simply reduced to a texture lookup in the lightmap.

The fixed directions should be uniformly distributed on the sphere to get a representative sample of the incident radiance. We use an iterative approximation of Poisson-Disc Sampling, similar to the algorithm described by [26]. The algorithm consists of an initially empty set of selected points and a set of candidate points:

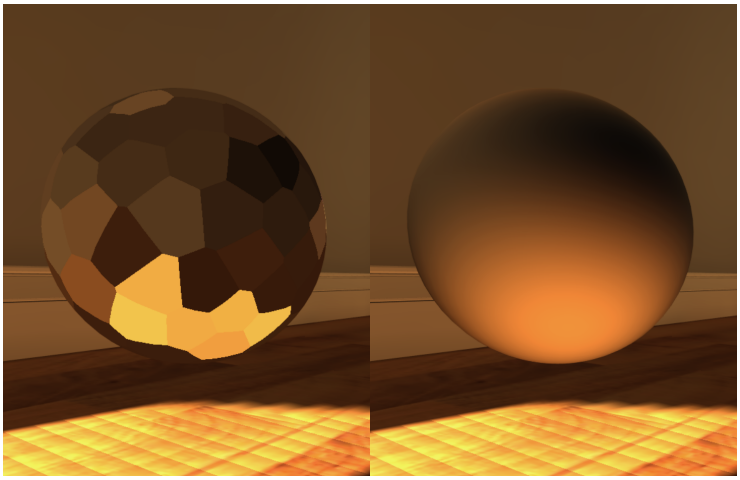

Figure 8: Left: Visualization of the radiance captured by a probe's relight rays, in 100 directions. Right: The resulting SH representation, using 16 coefficients.

1. Generate a new set of k independent candidate points on the unit sphere, each following a uniform distribution.

2. For each candidate point, calculate the minimum distance to previously selected points, and add the candidate with the greatest such minimum distance to the set of selected points. If the selected set is large enough, terminate. Otherwise, go to step 1.

The greater the parameter k is, the lower the risk of points being unnecessarily close.

5.7 Extracting the Global Illumination Solution

Finally, the global illumination solution at each receiver is computed by multiplying the coefficients in the light transport matrix with the corresponding coefficients of the probes. It is important to note that since there are only a few probes that contribute to each receiver we only need to compute and store the non-zero coefficients. The result is then stored in a separate lightmap. The scene can then be rendered using this lightmap in addition to the direct light.

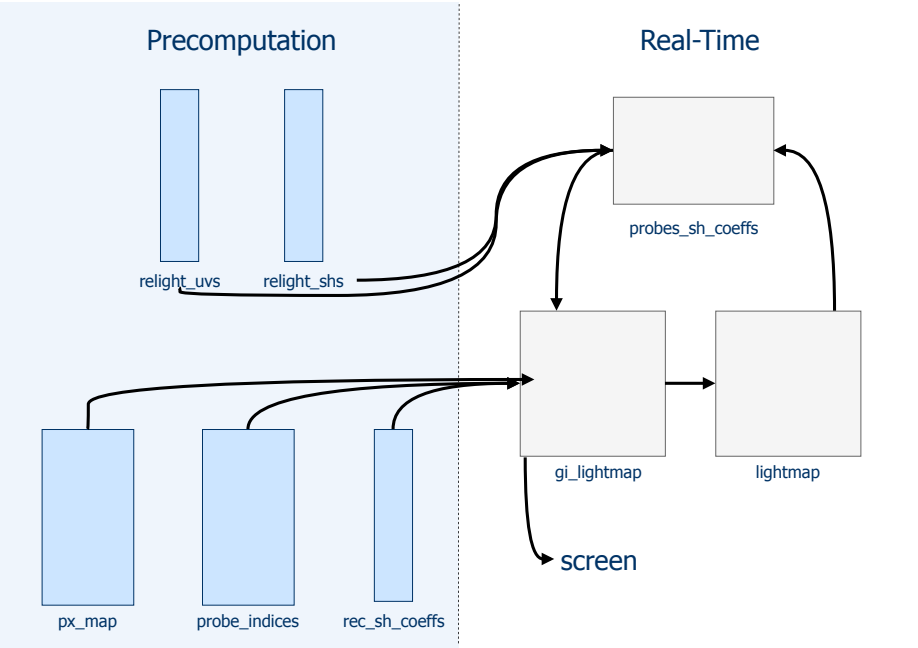

Figure 9: Diagram of the flow between the vertex buffers and textures used in the program.

5.8 Data Structures and Program Flow

The general flow of the program and the division between precomputation and real-time is illustrated in Figure 9, which includes the following textures and vertex buffers:

- `relight_uvs` – the uv coordinates that describe where the relight rays from each of the probes hit the scene, stored as a texture of RG32 floats (red = u, green = v) of size `num_relight_rays * num_probes`.
- `relight_shs` – the values of the included SH basis functions evaluated in each of the relight ray directions, stored as a texture of R32 floats of size `num_sh_coefficients * num_relight_rays`.
- `px_map` – the uv coordinates of each receiver, stored as a vertex buffer of size-2 integer vectors.
- `probe_indices` – the identifying indices of the eight closest probes to each receiver, stored compactly as a vertex buffer of size-4 integer vectors, with each integer (32 bits) storing two 16-bit values.

- `rec_sh_coeffs` – the local transport matrix, stored as a flattened-out texture of RGBA16 floats (each RGBA16 texel contains 4 SH coefficients, so that four consecutive texels together store the full 16 SH local transport coefficients of a certain probe in a certain receiver's list of its eight closest probes).

- `probes_sh_coeffs` (real-time) – the SH coefficients that represent the incident light field at each of the probes, stored as a texture of RGB16 floats (each color component has its own set of 16 SH coefficients) of size `num_sh_coefficients * num_probes`.

- `gi_lightmap` (real-time) – the light map for the resulting indirect light.

- `lightmap` (real-time) – the full light map, containing direct light + indirect light.

- `screen` (real-time) – contains the final render of the scene with direct light + indirect light; it is thus very similar to `lightmap`. The difference is that it only renders the surfaces seen from the camera's point of view, and computes the direct light per screen pixel rather than per lightmap texel. This results in a higher resolution, where only the (slowly varying) indirect light is limited by texture size and not the (fast varying) direct light.

The precomputed data is loaded at the start of the program and stored in textures and vertex buffers, and the `lightmap` texture initially only contains the direct light of the scene. The `relight_uvs` and `relight_shs` data are fed into a shader together with the current `lightmap` to compute `probes_sh_coeffs` – the SH coefficients that represent the incident light field at each of the probes. The main GI shader then combines this information about the radiance at the probes with the precomputed information about the scene geometry that is stored in `px_map`, `probe_indices`, and `rec_sh_coeffs`, resulting in the `gi_lightmap` texture, which now contains the illumination from the first bounce of indirect light. The sum of the direct light and the `gi_lightmap` is then used in two different ways: (1) to render the final image on the screen, (2) to update the `lightmap` texture. The second use (2) creates the final connection of a feedback loop between `probes_sh_coeffs`, `gi_lightmap` and `lightmap` as seen in Figure 9. After a single run through the loop, the `lightmap` has gone from including only the direct light to including direct light + the first bounce of indirect light. In the next frame, this new version of `lightmap` is used at the start of this cycle so that after a second run through the loop, the `lightmap` contains direct light + the first bounce + the second bounce of indirect light. This goes on indefinitely so that at the nth frame the n first bounces of indirect light are included in the approximation. It should be noted however that this typically converges after only a few frames (bounces), because every new bounce will contain less energy than the previous one.

5.9 Compressing the Local Transport Matrix

The previously discussed steps suffice to produce a good solution for the global illumination problem. However, to reduce the runtime memory footprint Silvennoinen and Lehtinen use clustered principal component analysis in line with the work of Sloan et al. [27]. That is, first the receivers are divided into clusters based on their position in world space, each of these clusters are then approximated using truncated singular value decomposition, tSVD [28, p.128].

We identify two problems with the clustering of receivers:

- it reduces the compressability by splitting the data into multiple smaller data sets
- it introduces discontinuities on cluster boundaries

Silvennoinen and Lehtinen point out the computational cost associated with computing the SVD directly. However, with randomized methods such as [29] this computational cost is negligible in comparison with computing the local transport matrix in the first place. There is, however, a very good reason to not compute the tSVD without first clustering the receivers. The local transport matrix T is very sparse. If it is approximated with truncated SVD $T \approx U\Sigma V^T$, U is generally a linear combination of all principal components. Since we only need to store the non-zero components of T, even if we remove a large number of principal components we will not get any compression as a result of losing the sparsity. By first applying clustering L is no longer sparse and the problem disappears.

Let us consider the actual optimization problem that we are trying to solve: let T be the local transportation matrix, P the vector of spherical harmonics for all probes, and $T^* = BD$ the approximation we are trying to find. The runtime computation that we perform is $B(DP)$. We want B to be as sparse as possible, so we want to minimize the l_0 pseudo norm, which counts the non-zero elements. The approximation also needs to be accurate. This leads to the formulation in Equation 14.

$$\begin{aligned} &\min |B|_0 \\ &\quad s.t. |T - BD|_2 < \epsilon \end{aligned} \tag{14}$$

where ϵ is the maximum allowed error, $B \in R^{nk}$, $D \in R^{km}$.

This problem is known as Dictionary Learning or the sparse coding problem. The dictionary learning problem has gathered a lot of attention during the last decade. It plays a large role in image processing problems such as denoising.

We will borrow some terminology commonly used in this problem formulation. k is known as the number of atoms, D is known as the Dictionary and B the sparse coding.

Dictionary learning is an NP-complete problem but there are many algorithms that present accurate approximations. Generally these algorithms iteratively solve two sub-problems:

1. Update the sparse codes given a constant dictionary

2. Update the dictionary given constant sparse codes

This formulation takes care of the sparsity problem of tSVD; however it reintroduces the problem of discontinuities seen with clustering.

If we add smoothness constraints to the solution we can decrease the discontinuities i.e. in the approximation the difference between spatially close samples should match the difference between the correct values of both samples. One approach to solve the smooth version of the problem can be found in [30], which solves the first sub problem with marginal regression second with Method of Optimal Coherence-Constrained Directions (MOCOD) [31].

Data: matrix to approximate X, dictionary D, sparsity parameter λ
Result: Sparse code B
$C \leftarrow D^T X$
for *i .. rows(X)* **do**
  $idx \leftarrow$ indices_of_largest_coeffs(C_i, λ)
  $\min_\beta |D\beta - X_{i,idx}|_2$
  $B_{i,idx} = \beta$
end

Algorithm 4: Marginal Regression

Where indices_of_largest_coeffs is a function that returns the indices of the n coefficients with largest magnitude such that the sum of those n coefficients are smaller than a sparsity parameter λ. We also use the notation of indexing multiple coefficients with a list of values such as that return by the previously mentioned function.

Since the solving for D is over-determined simply using least squares is reasonable. This is known as MOD or method of optimal directions. The previously mentioned method MOCOD is a modified version of MOD adding additional penalties to make the atoms more orthogonal and closer to normalized.

Data: matrix to approximate X, sparse code B
Result: Dictionary D
$\min_D |DB - X_{|2}$

Algorithm 5: Method of Optimal Directons

A simple dictionary learning algorithm can then be defined as

Data: matrix to approximate X, sparsity parameter λ
Result: Dictionary D, sparse code B
D $\leftarrow random\_matrix()$
while *not converged* **do**
  $B \leftarrow MarginalRegression(D, X, \lambda)$
  $D \leftarrow MOD(B, X)$
end

Algorithm 6: Simple Dictionary Learning

The idea introduced in [30] is to not only compute the correlation between the atoms and the sample to be approximated but instead the correlation between a neighbourhood of samples and the atoms. I.e., instead of computing $D^T X$ we compute $D^T XW$, where W is some weight matrix between the samples. This forces the method to use similar atoms for nearby samples which is desirable in our application where discontinuities negatively impacts visual fidelity.

Marginal regression is very fast compared to other methods such as solving the LASSO problem. For large dimensional problems it is approximately two orders of magnitude faster [32]. But our problem instances can be larger than what is feasible even for marginal regression. Therefore we introduce an addition to marginal regression that allows us to handle large dictionary sizes more efficiently. We call this method partial marginal regression.

Instead of calculating the full correlation matrix in each step we observe that the sparse coding for a given sample does not generally change that much between iterations. As such we allow each matrix to only be a linear combination of a small candidate set of the atoms. We define the candidate set as the atoms used in the previous iteration and the n atoms that are most correlated with each of the atoms used in the previous iteration.

Additionally instead of selecting the k coefficients such that their absolute sum is smaller than λ we simply select the λ' coefficients with the largest magnitude where $\lambda' \in \mathbb{N}$. This has the disadvantage of using more coefficients over all but the advantage that every sample is approximated with the same number of coefficients. This choice is made to simplify our run-time calculations. We have not investigated the impact of this decision.

An important implementation detail that holds for partial marginal regression as well as traditional marginal regression is that if we solve the minimization problem using the method of normal equations $min_x|Ax - y|_2 = (A^T A)^{-1} A^T y$ both $A^T A$ and $A^T y$ can quickly be computed from previous results. In each iteration we select the correct coefficients from $D^T D$ which is computed before the loop and $D^T X_{i,idx}$ can be computed by selecting the correct coefficients of the correlation vector and multiplying it with the length of the corresponding

Data: matrix to approximate X, dictionary D, sparsity parameter λ, previous sparse code B
Result: Sparse code B'
$DN \leftarrow$ rowwise_normalized(D)
$AC \leftarrow DN^T DN$
$ACS \leftarrow$
rowwise_select_s_largest_coeffs(AC)
for i .. *rows(X)* **do**
  $cand\_set \leftarrow$ indices_of_non_zeros($Bp.row(i)ACS$)
  $cand\_corr \leftarrow DN^T.rows(cand\_set)X.row(i)$
  idx $\leftarrow$ indices_of_n_largest_coeffs($cand\_corr$, λ)
  $\min_\beta |D\beta - X_{i,idx}|_2$
  $B'_{i,idx} = \beta$
end

Algorithm 7: Partial Marginal Regression

atoms. Additionally the inverse does not explicitly need to be computed, since $D^T D$ is semi positive definitive we can solve it using LDL decomposition. Since the matrix that is LDL decomposed is very small, [λ, λ], this operation is very fast.

5.10 Dynamic Objects

While the original formulation in [19] only supports static geometry and dynamic lights, we extend the algorithm to support dynamic objects by retrieving simply spatially interpolated irradiance from nearby probes to illuminate the dynamic objects. While simple spatial interpolation is less correct than the visibility-aware sheared version as has been shown in section 5.1, it is not dependent on any precomputed quantities other than those for the relight rays, and can therefore be evaluated at any position and direction in real-time. Note that the probes store radiance rather than irradiance. In the ordinary algorithm, irradiance is obtained through the numerical integration (with the cosine factor) in the precomputation of the local transport matrix. In our dynamic solution, we instead need to perform the conversion from the probes' stored radiance to irradiance purely in real-time. We do this using Ramamoorthi and Hanrahan's formulation [33, eq. 13], which combines the rotated clamped-cosine convolution with the evaluation in the normal direction. This is applied on all probes within the support radius r, and the evaluated irradiance from each probe is then simply spatially interpolated according to Equation 8.

This extension has a few inherent limitations; because the relight ray hit points are still precomputed (and based only on the static geometry) the radiance stored at the probes at run-time is independent of the dynamic objects, i.e.

the dynamic objects are invisible to the probes. This means that the dynamic objects can receive indirect light from the static environment but not from each other, and they can also not cast indirect light (nor indirect shadows) onto the static environment. What they *can* still do however is to cast *direct* shadows (by blocking direct light) onto the static environment, which in turn has an effect on the indirect light on both static and dynamic geometry.

6 Technique 3: Light Propagation Volumes

Light propagation volumes (**LPV**) is a technique for global illumination developed by Crytek, first described in 2009 by Anton Kaplanyan [20]. The algorithm operates fully in real-time and uses no precomputed steps. Light propagation volumes as implemented in this paper is able to approximate the first bounce of light coming from direct light sources by using the surfaces lit as secondary (low-frequency) light sources. Our implementation of the algorithm is also inspired by the Cascaded Light Propagation Volumes [34] paper; published in 2010 by Anton Kaplanyan and Carsten Dachsbacker. Our version alters the original implementations presented by Crytek. The reason for this being the current day restrictions with the WebGL API where some features of the modern versions of OpenGL are not yet supported by WebGL.

The general idea behind light propagation volumes is to use the surfaces lit by direct light as secondary light sources to calculate one bounce of indirect light. The algorithm is divided into 4 steps which are described in detail in the sections below. The first step generates surface data and stores it into a reflective shadow map (RSM, section 2.9). The RSM data is then used to inject the surface data into a 3D grid of virtual point lights. We also use geometry injection to add blocking of light and indirect shadows during this step. The light injected into the grid is then propagated between cells to achieve a better light spread and light bleeding. Finally, in the fourth and final step we calculate the indirect light and use it when rendering the scene.

6.1 Reflective Shadow Map Generation

The first step of the light propagation volume algorithm is to generate a RSM for every light source of the scene. This is useful in later steps as it allows for efficient data lookup with little computation needed.

Our implementation of light propagation volumes uses a 4096×4096 simple shadow map for shadows. However it is not feasible to use a reflective shadow map of this size in the injection stage, as it would require large amounts of computing power and thus be too slow. To speed up the process, the RSM is downsampled to a resolution of 512×512 by rendering it to another framebuffer using a smaller viewport before initiating the injection stage.

6.2 Radiance Injection

After rendering the reflective shadow map, a point cloud bounded by the width and height of the RSM is rendered as single vertices. Thus, we have a 2D point

cloud containing points with the positions $0 \leq x, y \leq RSM_s$, where RSM_s is the RSM size. The idea is to represent every texel in the RSM with a point in the point cloud. The point cloud is then passed on to the vertex shader where every vertex is automatically assigned a vertex ID. The ID can easily be fetched using the built in variable $gl\_VertexID$. The ID id_p of a point is then transformed into a texel t_{xy} on the RSM (Equation 15).

$$t_{xy} = (id_p \% RSM_s, id_p / RSM_s) \tag{15}$$

where % denotes the modulo operator.

The texel from each of the respective RSM textures is then fetched using t_{xy}. To avoid self illumination, every world space position is displaced by half a cell in the direction of the surface normal.

The objective of the injection stage is to save the lighting contributions in different directions for the different positions as virtual point lights in a grid in a 3D-texture. Using the grid size, cell size, and the displaced world space position fetched from the RSM, it is possible to calculate a position in the grid for a virtual point light. The grid cell is fetched by dividing the distance from the minimum grid cell to the displaced world space position by the cell size.

In modern OpenGL versions you normally render a 3D-texture slice by slice, meaning that you render every depth layer as an ordinary 2D-texture and stack them upon each other. When rendering 3D-textures in modern OpenGL the depth layer selection during runtime is handled via the geometry shader's built-in variable gl_Layer. Unfortunately, the current WebGL pipeline has no support for geometry shaders. This means that there is no simple way of selecting the depth layer to render to during run-time. To solve this, a 2D-texture is utilized instead by stacking the depth layers side by side. This means that a grid that would be of the size $32 \times 32 \times 32$ instead could be represented as a 2D-texture of size 1024×32.

After the 3D grid location has been computed the position on the texture to render to need to be calculated. The 3D grid location gl_{xyz} and the grid size g_s are used to find the texture coordinates t_{xy} (Equation 16):

$$t_{xy} = (gl_x + gl_z g_s, gl_y) \tag{16}$$

The texture coordinate is then transformed into normalized device coordinate (NDC) [35] space (where $-1 \leq x, y, z \leq 1$) and used as the rendering position output by the vertex shader. The previously fetched RSM texel is also sent through to the fragment shader.

In our implementation we use spherical harmonics of the second band, which means that 4 coefficients are used. The directional distribution of a color is represented as spherical harmonics projected on a clamped cosine lobe centered around the surface normal. The cosine lobe is then scaled by the scalar intensity of each of the colors (RGB) to get the directional intensity of a color. The coefficients (c_0, c_1, c_2, c_3) for the cosine lobe around a normal $n = (x, y, z)$ are as shown in Equation 17 [36]:

$$\begin{aligned} c_0 &= \frac{\sqrt{\pi}}{2} \\ c_1 &= -\sqrt{\frac{\pi}{3}}y \\ c_2 &= \sqrt{\frac{\pi}{3}}z \\ c_3 &= -\sqrt{\frac{\pi}{3}}x \end{aligned} \tag{17}$$

This vector is then renormalized to form a hemispherical lobe by dividing the resulting vector (c_0, c_1, c_2, c_3) by π. Additive blending is used in the injection stage. Since positions inside a given grid cell are not accounted for, if there are multiple texels that translate into the same grid cell the light is added to that grid cell multiple times. Therefore, the weight of each texel has to be accounted for when adding it to the cell. Each cosine lobe is therefore scaled by a texel weight calculated by dividing the grid size by the RSM size.

As described above, the cosine lobes are stored in three separate vectors, each scaled by the intensity of their respective color channel. The scaled cosine lobes are then rendered into three different grid textures; one for each color channel. The resulting grid after injection is visualized in Figure 10.

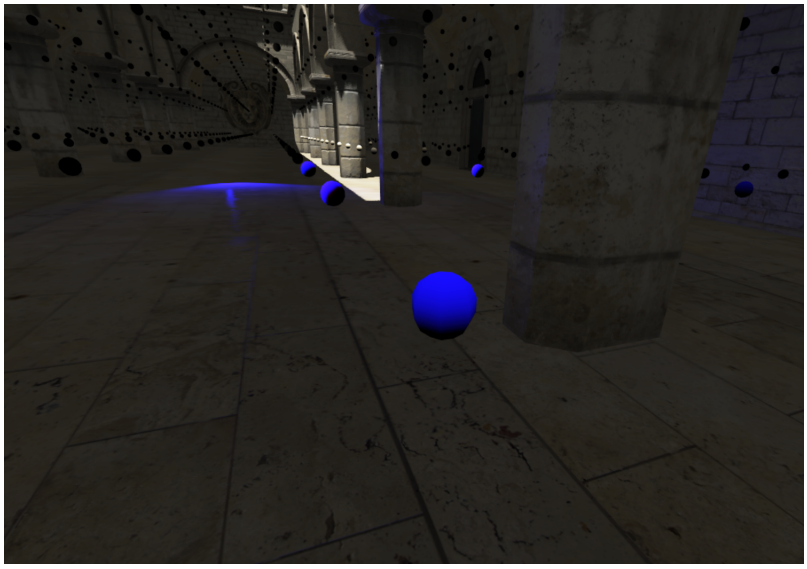

Figure 10: Spherical harmonics in the grid after light injection

6.3 Geometry Injection

In addition to performing radiance injection a geometry injection step is also performed, where a volumetric approximation of the scene's geometry is stored in a second grid (geometry volume). These approximations are used to block the light during propagation which results in the computation of indirect shadows. The geometry volume (**GV**) is displaced such that the center of the geometry volume lies on a corner of the light propagation volume. In the geometry injection step the spherical harmonics projections of the blocking potential are injected into the geometry volume which is used during the propagation step as mentioned in Section 6.4.1.

6.4 Radiance Propagation

As previously mentioned and as the name implies the purpose of the propagation stage is to spread the light of the cells to its neighbours in order to give a better light distribution within the grid. The reason for this is that the light would be visible with very sharp edges which then would seem unnatural if propagation is not applied. This can clearly be seen in Figure 11. By propagating the light the light bleeding effect, where the color of a surface hit by direct light affects surfaces hit by the indirect light bounce, is achieved.

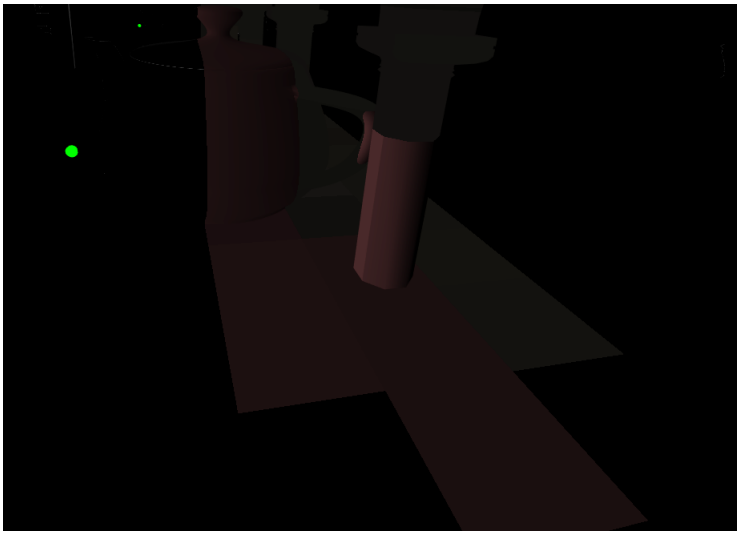

Figure 11: Light before propagation is applied. The edges of the grid cells are clearly visible giving the light an unnatural appearance.

The propagation step consists of n sequential iterations where n should be adjusted to each application and scene in order to optimize the result. In the propagation step the light contribution from every cell's 6 neighbouring cells (in 3D) are sampled. A 2D illustration with 4 neighbours can be seen in Figure 13 to the left. After sampling the neighbours' contributions the sum of these are written to the cell's own contribution. To compute the contribution of neighbour n, the integral of the outgoing radiance going through each face f of n is calculated, as seen in Figure 13 to the right. The sum of these makes up the contribution of neighbour n. The four side faces of a cell can be seen in Figure 12.

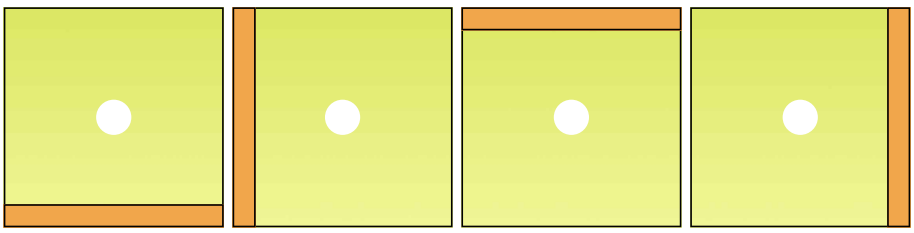

Figure 12: The four side faces of a cell, in orange.

The input for the initial iteration of the propagation step is the LPV from the injection step. This LPV is then modified in each propagation iteration and sent to the next iteration. After n number of iterations the resulting LPV is used in

scene lightning, see section 6.5. Each cell in the LPV stores the intensity as a SH-vector, one for each color channel (RGB).

The light is propagated from the center of the source cell in direction ω_c towards the center of each face in the neighbouring cell. This is done for all 6 neighbouring cells. The intensity, $I(\omega_c)$, is evaluated by taking the dot product of the sampled SH coefficients of the neighbour and the SH projection of the direction vector ω_c of the visibility cone.

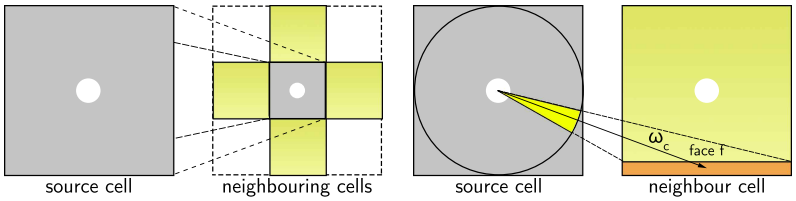

Figure 13: Left: Each cell in the LPV stores the directional intensity used to compute the light that is propagated from a source cell to its 4 neighbors (2D), 6 in 3D. Right: The flux is computed through each face f of the neighbour cell.

To avoid inaccurate integral values stemming from our low-order SH approximations the subtended solid angles of each face in the neighbour cell are computed. The solid angles used in our implementation are $S_d = 0.4006696846$ for the direct face (back face) and $S_s = 0.4234413544$ for the side faces. The flux is the computed as $\phi_c = S_d/\pi \cdot I(\omega_c)$.

The incoming flux is also re-projected to the center of the cell by determining a re-projection direction r_c. A cosine lobe of r_c is projected into SH coefficients and multiplied by the flux $\phi_i = S_s/\pi \cdot I(\omega_c)$. This is done for each face f of the neighbouring cell n. Pseudo code of the propagation algorithm can be found in Algorithm 8. The steps are repeated for every color component.

6.4.1 Blocking of Light

In the propagation step we also include the blocking of light which is computed from the geometry volume. The spherical harmonic coefficients of the geometry volumes are interpolated and the occlusion for each color channel in the propagation direction is evaluated. This is done to attenuate the intensity of the light and produce indirect shadows. This evaluation of the occlusion is not performed in the first step of propagation after radiance injection in order to prevent self-shadowing.

Data: Direct face solid angle S_d, Side face solid angle S_s, neighbours of source cell $neighbours$, faces of a cell $faces$
Result: Contribution C_s of source cell
for *neighbour n in neighbours* **do**
  if *!first iteration* **then**
    evaluate occluded contribution c_d of direct face
  end
  $c_n \leftarrow c_n + c_d \cdot I(\omega_c)$
  for *face f in faces* **do**
    if *!first iteration* **then**
      evaluate occluded contribution c_s of side face
    end
    $c_n \leftarrow c_n + c_s \cdot I(\omega_i)$
  end
end
$C_s = c_n$

Algorithm 8: Radiance Propagation

6.5 Scene Lighting

After the propagation step is finished, the grid has been populated with indirect light that has been distributed throughout the grid and projected onto the faces of the grid cells. What remains is to use this data to light the scene.

To evaluate the light of a surface fragment the position of the fragment in the grid is first calculated. When fetching the light contribution of the grid cell from the grid, trilinear filtering is performed to achieve smooth light transitions. Because of our 2D representation of the 3D grid we need do the trilinear filtering manually in the fragment shader. Leveraging the hardware bilinear filtering, only 2 texture lookups are needed for every value fetched. Since one grid is used for every color channel, 3 values have to be fetched, meaning 3×2 texture lookups need to be performed.

The fragment's normal $n = (xyz)$ is then projected into spherical harmonic basis to form the resulting vector $C_n = (c_0, c_1, c_2, c_3)$ by multiplying n by the first two bands of spherical harmonic coefficients (Equation 18) [11].

$$
\begin{aligned}
c_0 &= \frac{1}{2\sqrt{\pi}} \\
c_1 &= -\frac{\sqrt{3}}{2\sqrt{\pi}}y \\
c_2 &= \frac{\sqrt{3}}{2\sqrt{\pi}}z \\
c_3 &= \frac{\sqrt{3}}{2\sqrt{\pi}}x
\end{aligned}
\tag{18}
$$

The projected normal C_n is then dotted with each of the trilinearly interpolated values Gr, Gg, Gb fetched from the grid earlier to form the resulting vector I_{rgb} containing the indirect light irradiance as shown in Equation 19.

$$I_{rgb} = (C_n \cdot Gr, C_n \cdot Gg, C_n \cdot Gb) \tag{19}$$

As negative light contributions in any channel is invalid, the vector I_{rgb} is clamped to values above 0 and divided by π to renormalize it [37]. The irradiance vector is then multiplied by the diffuse color of the fragment to be lit, resulting in the diffuse indirect light ID_{rgb} for the fragment.

6.6 Dynamic objects

Even though the light propagation volumes technique is capable of running completely in real time with dynamic lights, camera, and geometry it is not necessary to update the light grid in every frame. When rendering a fully static scene only the scene lighting step runs each frame, using the previously stored light grid during rendering. Only when something has been updated since the last rendered frame the light grid has to be updated, thus saving computation time which can be very beneficial.

7 Results

We present the individual results of the techniques below, follow by a side by side presentation. All results are rendered in Windows 10 by Mozilla Firefox (version 59.0.3) on a 1920×1080 resolution, using an Intel i7-6700k with an Nvidia Geforce GTX 780 (3 GB VRAM) and 16 GB of RAM if nothing else is mentioned.

For testing and evaluation purposes LIVING ROOM [9] and a modified[1] CRYTEK SPONZA [9] was used (from now on referred to as SPONZA).

7.1 Technique 1

Unless otherwise mentioned, all images for this technique are rendered with cubemaps of the size 6×256^2 pixels, octahedrals of the size 1024^2 pixels, and the irradiance and filtered distance of the size 128^2 pixels. For filtering, 2048 and 128 samples per pixel (**SPP**) were used, respectively.

For glossy indirect light, i.e. reflections, we did not manage to achieve results close to the ones presented in the original paper (compare to [18]). While we did use the supplementary code provided for the ray-tracing, changes had to be done, as described in section 4.4. In Figure 14 the full technique is shown, i.e. with both diffuse and glossy indirect light. From the figure it is clear that the quality of the reflections is not very good. Large portions of the screen could not successfully be ray-traced, but even the parts that do work seem to be of low resolution: notice the jagged lines on the reflections of the arches.

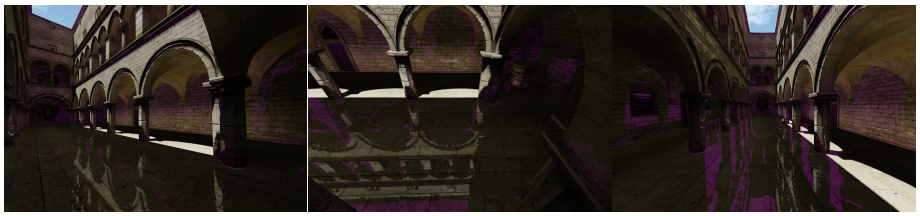

Figure 14: Direct light, diffuse and glossy indirect light. While glossy indirect light does work it is not of sufficient quality. Glossy indirect light due to rays that could not be successfully traced (UNRESOLVABLE or MISS after tracing is done) is colored magenta to highlight the problem.

While the number of samples and s (in Algorithm 3) for the filtered distance had little significance to the overall quality of the GI, we found that the number of samples for the irradiance was closely tied to the quality. In a tradeoff between

[1] Curtains and plants were removed and a green TEAPOT model [9] was inserted for color.

performance and quality we found 2048 SPP for a 128^2 irradiance map worked best, since it allowed good quality indirect light at real-time performance. With 2048 SPP there is still a significant amount of high-frequency noise in the irradiance maps, which is not ideal. To remove most of the high-frequency artifacts, SPP in the 10^4 order of magnitude or greater is needed. In Figure 15 images rendered with only direct and diffuse indirect light can be seen; some artifacts can be observed in the LIVING ROOM scene on surfaces with smoothly varying normals, such as the sofa cushions on the bottom right picture.

Figure 15: Direct and diffuse indirect light only, rendered using technique 1. Top row: SPONZA scene [9] with $8 \times 8 \times 4 = 256$ probes. Bottom row: LIVING ROOM scene [9] with $3 \times 3 \times 4 = 36$ probes.

For both glossy and diffuse indirect light we found the probe count and placement to be closely tied to the quality of the GI (Figure 16). The probe count is also directly tied to the precomputation time. Since we only precompute a fixed number of probes per frame (we found 2 probes per frame worked best for most computers) and spread out the precomputations for probes over multiple frames, the number of probes changes the turnaround time from a change to correct GI respecting the new state. Given $4^3 = 64$ probes, 2 probes per frame, and that 60 FPS can be maintained, the turnaround time would be 0.53 seconds.

Figure 16: Direct and diffuse indirect light only, rendered using technique 1. The probe locations are drawn as green spheres in the scene. It can be observed from the images that probe placement has a significant impact on the quality of the GI. In the left image a probe is located right behind the sofa; because the bright windows cannot be seen from that location the area is very dark. In the right image there is no probe at that location, so it is significantly brighter behind the sofa.

7.2 Technique 2

7.2.1 Effect of varying the number of SH bands

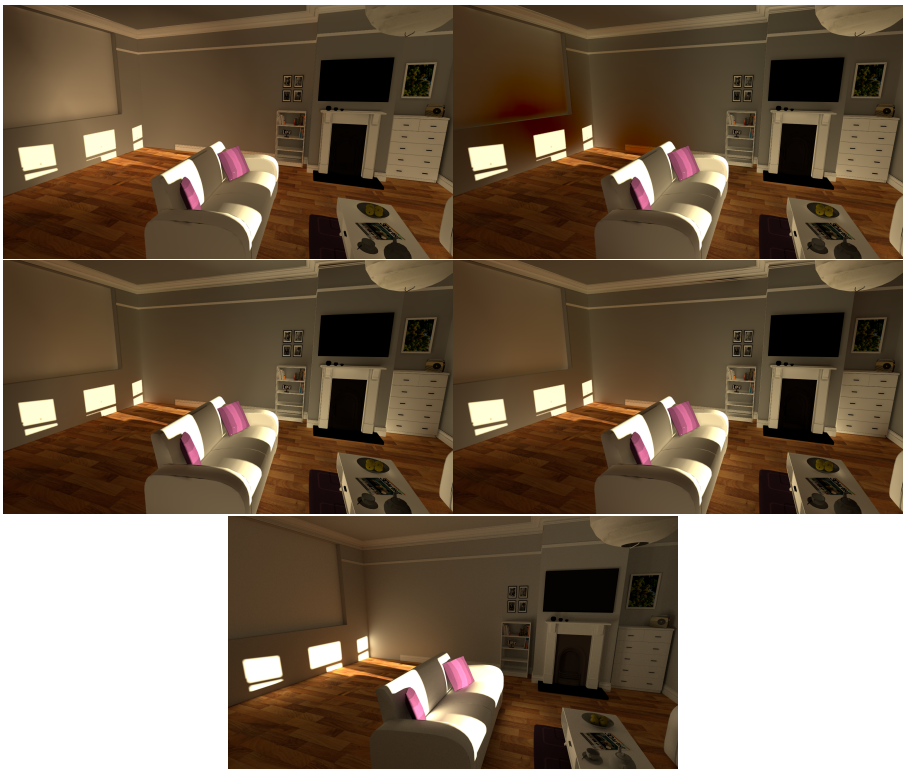

Figure 17: Rendering the scene with different number of bands of the spherical harmonics. In order top to bottom, left to right. 1, 4, 9, 16 coefficients respectively. Reference at the bottom.

In Figure 17 we can see that if we increase the number of bands that are used in spherical harmonics approximation the image quality generally increases. Note especially the light in the center of the sofa, the ambient occlusion in the niche on the right hand side and the slight increase in brightness inside of the fire place. Note also the decreased quality with four coefficients compared to with only one.

The precomputation time for the LIVING ROOM scene with 64 coefficients was 58 minutes on a Toshiba Satelite L50-B laptop from 2014, 2.6GHz quad-core i7, integrated graphics card with 2GB VRAM. For the SPONZA scene the precomputation time was similar. Precomputing only 16 coefficients reduces the precomputaton time by roughly a factor of two.

7.2.2 Dynamic objects

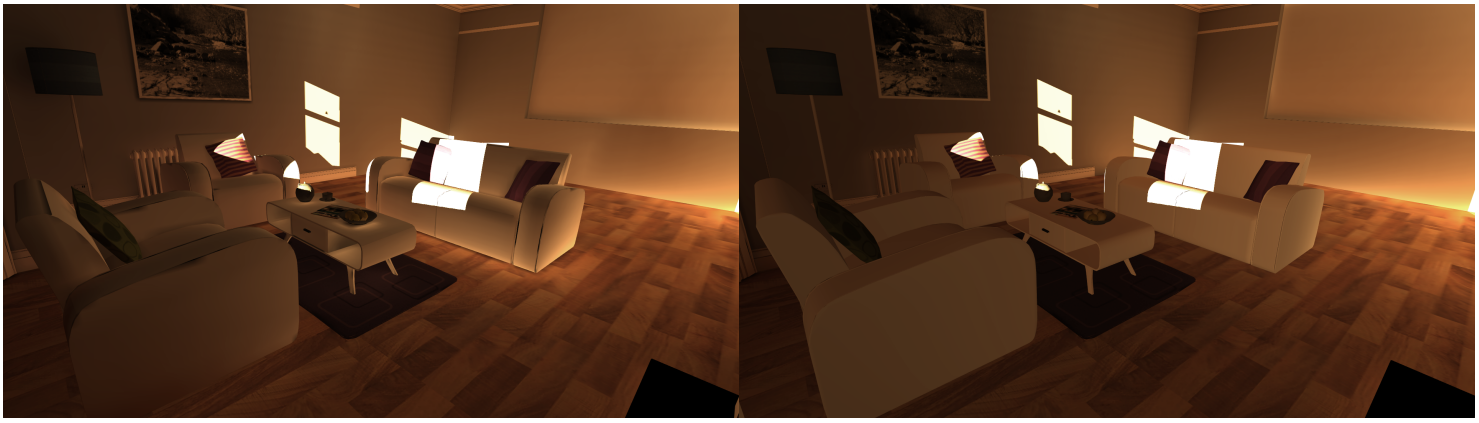

Figure 18: Left: LIVING ROOM scene with both the room and the interiors statically precomputed (using 15 probes). Right: LIVING ROOM scene with the room statically precomputed (using 15 probes), but the interiors fully dynamic.

Our extension to include dynamic objects seems to work surprisingly well in many cases with the dynamic objects relatively seamlessly blending in with the static background (see Figure 18). In the above example it even performs *better* in one way: it gets rid of the artifacts caused by the lightmap texture's limited size (the black spots). At the same time it clearly has a flatter, less detailed appearance than the fully static version.

7.3 Technique 3

The light propagation volumes technique allows for fully real time rendered scenes. Unless otherwise mentioned a grid size of 32^3 and RSM size of 512^2 is used with 16 propagation iterations on the Sponza [9] scene. When presenting the results for technique 3 we will refer to dynamic and static scenes. In dynamic scenes we update the shadow map for every light source between every frame, while in static scenes we do not.

7.3.1 Propagation Iterations

We found that the number of propagation iterations and the grid size used when rendering had significant impact on the quality of the indirect light. The grid has to cover the entire area of which the indirect light calculations should be done, otherwise the light will look irregular and unrealistic. For most scenes we found that $\frac{n}{2}$ propagation iterations, where n is the grid size, gave us sufficiently good results with good performance, as displayed in Figure 19.

Figure 19: From left to right: Sponza rendered without indirect light, 8 propagation iterations, 16 propagation iterations. All without ambient light. Many dark spots that should be lit are still visible when using 8 propagation iterations.

7.3.2 Many Lights

Technique 3 puts a lot of emphasis on performance and scales well with numerous light sources in a static scene, albeit not very well in dynamic scenes. When using 5 spotlights in a dynamic scene we still manage to get as high as 55 frames per second with a frame time of 18.18 ms, this diminishes very quickly when raising the amount of light sources until reaching 10 light sources, when the frame rate decline begins to stagnate, as shown in Table 1. In addition to the spotlights, the results below are rendered with one directional light.

Sponza (dynamic)		
Spotlights	FPS	Frame time
0	163	6.13
5	55	18.18
10	14	71.43
15	14	71.43
20	11	90.90
25	9	111.11

Table 1: Performance comparison of dynamic scene performance using multiple spotlights measured in frames per second (FPS) and milliseconds for the Sponza scene.

When rendering static scenes there is little overhead and our implementation scales very well with a high number of light sources. We can render a static Sponza, using 100 spotlights in 50 frames per second. The results for static scenes are presented in Table 2

Sponza (static)		
Spotlights	FPS	Frame time
0	410	2.44
50	140	7.14
100	50	20.00
150	34	29.41
200	24	41.66
250	20	50

Table 2: Performance comparison of static scene performance using multiple spotlights measured in frames per second (FPS) and milliseconds for the Sponza scene.

7.3.3 Incorrect Light Bleeding

As described in section 6.2, the light in the cells are displaced half a cell in the direction of the surface normal to prevent light bleeding through surfaces. It is still a fact that some surfaces are thin enough to be completely surrounded by grid cells and when we do trilinear filtering there is a chance that we fetch a value on the wrong side of a surface. Because of this, even though effort has

been made to avoid it, light bleeding through thin surfaces is still a problem in our implementation, Figure 20 visualizes this problem.

Figure 20: Image rendered with significant increase in amount of indirect light. While not obvious, you can still see the green light bleeding from the teapot on the lower level.

7.4 Comparative

As discussed in section 4.2 technique 1 allows for both diffuse and glossy reflections through the probe image-space ray-tracing. For the sake of fairness, though, it will be compared to the other techniques with only diffuse indirect light since that is supported by every technique.

Figures 21 and 22 show a visual comparison between the different techniques and ground truth for two different scenes. For the Sponza scene very convincing result can be achieved using technique 1 and 2, compared to the ground truth. Technique 3 also provides good results, but compared to the ground truth there are some significant differences, such as the amount and distribution of indirect light on the ground and back wall in Figure 21. For technique 2 some spot-like artifacts can be seen near the teapot. For comparison an ambient GI approximation was also included. For ambient light a small factor of the material color is simply added to the scene uniformly. While at first glance it might look convincing for this scene it also lacks any form of color bleeding and all unlit surfaces are equally bright; take notice of the brightness under the arches.

For the Living room scene the results differ more. Technique 2 achieves very accurate GI compared to the ground truth, while technique 3 looks too bright and washed out throughout and technique 1 looks too dark on some surfaces. Technique 1 especially suffers on surfaces where indirect light of more than one

bounce is visually significant, such as the left wall, behind the sofa. A significant difference can also be seen comparing all of the three techniques to the ambient approximation. Since the color of the incoming light and the floor is tinted towards yellow the ambient reference looks wrong since it does not take that into account.

Run-time performance comparisons can be seen in Tables 3 and 4. At idle all techniques performs well and are able to maintain 60 FPS (i.e. frame time < 16.67 ms). Techniques 2 and 3 are also able to maintain 60 FPS on change, with only minor impacts on frame time. Technique 1, though, has significant performance degradation on change and for Living room is not able to maintain 60 FPS. For the performance comparisons technique 1 precomputes two probes per frame on change. It is possible to maintain 60 FPS if only one probe is precomputed per frame, but this also doubles the turnaround time.

The precomputation time of technique 1 (i.e. turnaround time) is limited to a couple of seconds, and increases linearly with the number of probes. Technique 2 has a significantly longer precomputation time, ranging from minutes to hours depending on the scene. A larger scene will have more receivers and take longer to precompute. Technique 3 needs no precomputation.

In technique 1, for objects or light to change with correct global illumination, the probes has to be precomputed for each frame. Technique 2 does not fully support moving objects but light can move if the lightmap is updated and the global illumination recalculated for each frame. Technique 3 fully supports dynamic light and objects. However, the reflective shadow map of each light has to be redrawn at each frame when using dynamic scenes, so performance scales with the number of lights.

Sponza

Technique	Frame time (Idle)	Frame time (On change)
1	4.78	13.70
2	2.47	4.57
3	2.44	6.13

Table 3: Performance comparisons for the different techniques, measured in milliseconds, for the Sponza scene [9].

Living room

Technique	Frame time (Idle)	Frame time (On change)
1	5.81	21.28
2	1.86	4.17
3	3.50	6.71

Table 4: Performance comparisons for the different techniques, measured in milliseconds, for the Living room scene [9].

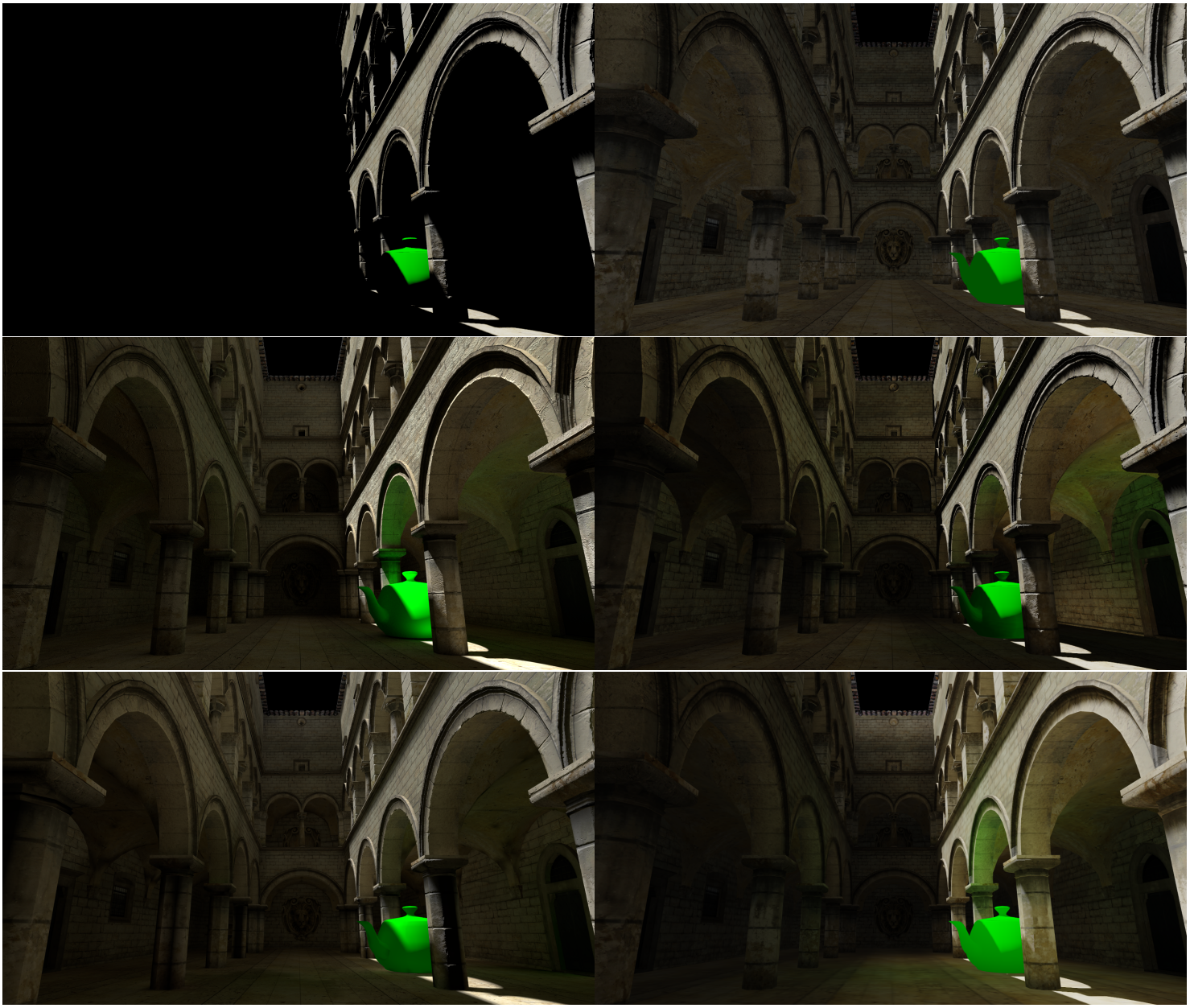

Figure 21: Comparison between only direct light, direct and ambient light, the three techniques, and ground truth for the Sponza scene [9]. Top left: only direct light. Top right: direct and ambient light. Middle left: path traced ground truth using Blender Cycles Renderer. Middle right: technique 1. Bottom left: technique 2. Bottom right: technique 3.

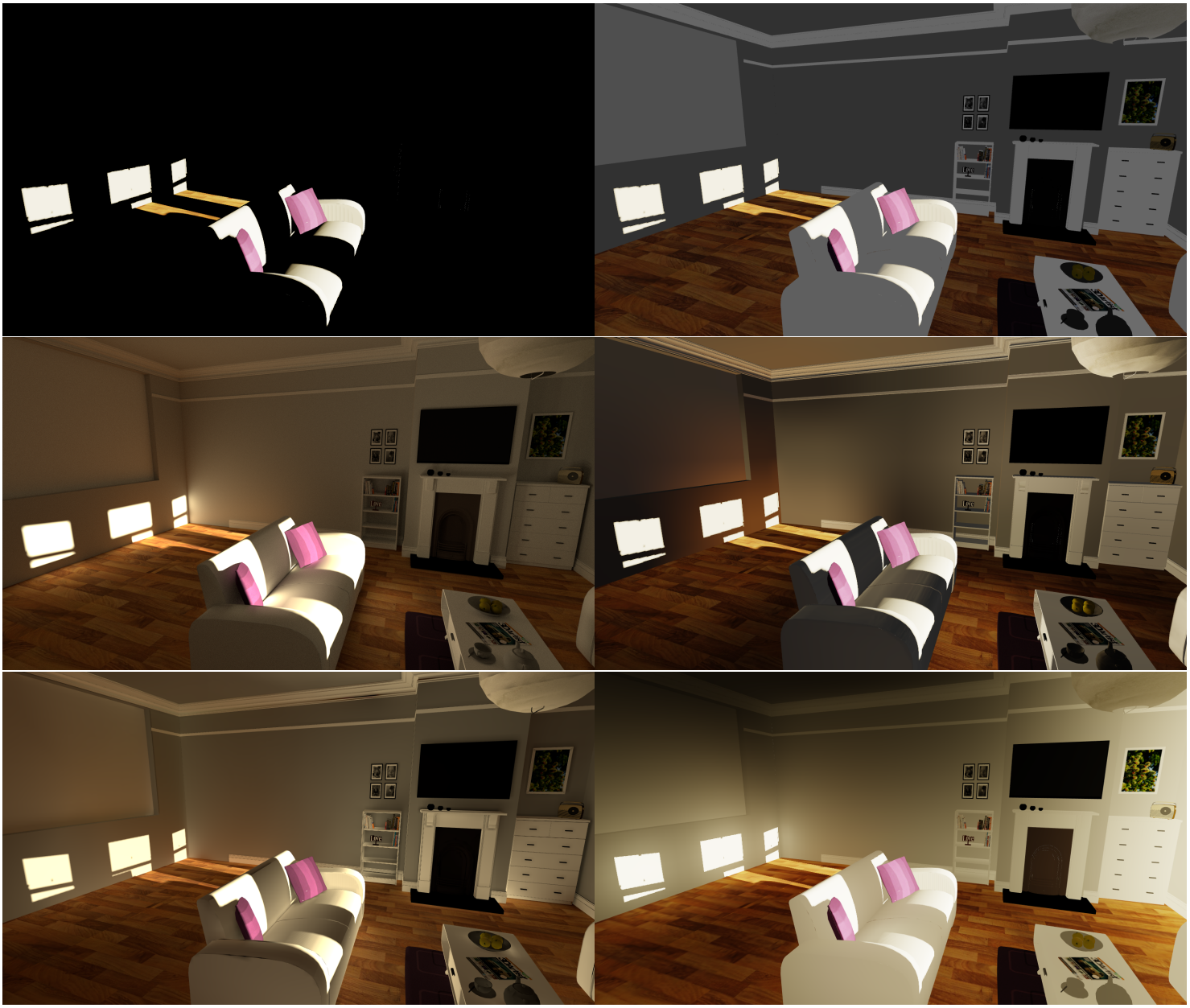

Figure 22: Comparison between only direct light, direct and ambient light, the three techniques, and ground truth for the LIVING ROOM scene [9]. Top left: only direct light. Top right: direct and ambient light. Middle left: path traced ground truth using Blender Cycles Renderer. Middle right: technique 1. Bottom left: technique 2. Bottom right: technique 3.

8 Discussion & Future Work

8.1 Technique 1

The Precomputed Light Field Probes technique [18] fulfills our aim reasonably well for diffuse indirect light. The illumination is reasonably realistic and when only calculating the diffuse light and omitting the ray-tracing necessary for specular reflections, the performance is very good as well. The big drawback is the precomputations needed, which makes dynamic objects a problem. However, the average precomputation time for a modern desktop computer is less than a millisecond per probe, which makes precomputation on the fly when the scene changes fairly seamless, unless a lot of lighting and objects change quickly or the scene requires a very large number of probes. There are also factors that can be changes to affect this. If the context is a highly dynamic environment, the number of probes and the resolution of the environment maps could be reduced drastically so that precomputation can be done in real-time. If the scene is largely static and it is acceptable for the global illumination to "snap into place" shortly after something is moved or changed, a higher number of probes with high resolution maps can be chosen.

Regarding the glossy reflections we believe the suboptimal results to be the result of our rewrite, as discussed in section 4.4, not being perfect. When rewriting the code we tried to achieve a perfect one-to-one mapping, functionality wise, but it is very possible that this was not successful. In any case, the glossy reflections are also very costly for complex scene [18, tab. 1] and are not relevant for the comparison in this paper.

As mentioned in section 7.1 there exist high-frequency noise in the 128^2 irradiance maps after filtering with 2048 SPP. To maintain high enough performance to be able to precompute in the background it is realistically not possible to go much higher than 2048 SPP, and definitely not as high as 10^4 SPP as would be required. By lowering the resolution of the irradiance map fewer SPP would be required, but it would also not produce as smooth results. One possible way of achieving smooth results could be to encode the irradiance into spherical harmonics instead of an explicit irradiance map. Spherical harmonics guarantee low-frequency changes while allowing infinite irradiance resolutions. This would also require less VRAM than the current solution.

Also mentioned in the results is how the probe placement has an effect on the quality of the GI. In our implementation we replicated what was done in the original paper [18] and placed probes in an axis-aligned grid with fixed steps between probes. While it allows for very optimized shader code for the ray-tracing it is not a strict requirement. It is conceivable that a smart probe placement strategy, such as the one used in technique 2 for receivers, could

be employed. Combined with a spatial data structure such as a kD-tree for picking the n closest probes to a point we hypothesize that similar real-time performance could be achieved.

8.2 Technique 2

It is important to discuss the places where our implementation diverges from that of Silvennoinen and Lehtinen. We changed how the probes are interpolated as described in 5.4, this does not seem to negatively impact visual fiedelity but only improve on the time needed for precomputation. We experimented with another approach for compressing the local transport matrix which meant that we did not have the time to implement the originally described method. Additionally there are a number of places where the approach is not described in enough detail for us to follow the paper.

- If, and in that case how, the lightmap is padded or if other approaches to mitigate lightmap seams are used
- If it uses raytracing or rasterization for computing the local transport matrix
- If any form of windowing was performed on the spherical harmonics to reduce ringing artifacts

Additionally our results only use the first 4 bands – 16 coefficients – of the spherical harmonics while Silvennoinen and Lehtinen generally use 8 bands – 64 coefficients. This reduces visual fidelity but improves performance.

The use of dictionary learning instead of clustered singular value decomposition for the compression of the local transport matrix did not quite yield the results that we were hoping for. While the problem formulation is quite accurate it does not take into account the need to store indices for sparse representations which for a clustered approach only need to be stored once per cluster. In addition as the scene size grows larger the local transport matrix grows more and more sparse, clustered approaches naturally divides large scenes into smaller ones and as such the increased sparsity does not cause any problems. However, for dictionary learning the increasing sparsity of the local transport matrix both increases the computational cost and decreases the resulting quality. Our augmentations to traditional marginal regression does mitigate these problems and for small scenes it does produce good results, however, for large scenes the results are still lacking.

While there was not enough time to explore other representations for evaluating the relight rays than the lightmap and uv approach described in 5.6 other alternatives are possible. The simplest of which is to store the world space position,

the normal and the color of the surface where the relight ray hit. Rendering the lightmap adds an additional pass where the entire scene must be rendered. Additionally it introduces another approximation since the lightmap has finite resolution. However, it only requires minimal additional memory, is easy to implement and has constant time complexity with regards to the number of relight rays. Depending on how many bands of the spherical harmonics are used to approximate the probes and consequently the number of relight rays necessary, diverging from the lightmap + uv representation might be possible.

8.3 Technique 3

Light propagation volumes is a great method of approximating global illumination in real time with many advantages, such as the flexibility; it is possible to use this technique in many different scenes with only minor configurations. But the technique also has its drawbacks, such as the incorrect light bleeding discussed in section 7.3.3 or the fact that it only supports low frequency indirect lighting[20]. The technique is also only able to approximate the first bounce of indirect light. It can be extended to handle multiple bounces but these methods has their drawbacks [34]. There are also extensions to LPVs that allow for glossy reflections or participating media lighting[20]. Despite this our implementation of LPVs achieves a reasonable result with respect to image fidelity and realism but the strength of it lies in its ability to handle global illumination for fully dynamic scenes in real-time.

8.3.1 Many Lights

In our implementation we use spotlights with positions generated on the CPU that are sent to the GPU for light calculations. We update every one of the shadow maps when the scene has been changed. An update of every single shadow map is not necessary when conditions might only have changed for one of the light sources. We would have liked to optimize this adding a scheme where we keep track of individual light sources and thereby only update the ones whose condition changes.

Another way of simulation a large number of light sources is by injecting already propagated point lights in the grid. This method could be done on the GPU without the overhead we have in our current implementation. This method is discussed in the original light propagation volumes paper and reach great results with lighting calculations taking 16.5 ms with 3000 light sources[20].

8.3.2 WebGL Limitations

As discussed in section 6.2, we use a 2D texture to represent a 3D grid due to the limitations of current day WebGL. When fetching a value from the 3D texture position (x, y, z), we instead fetch it from $(x+(z \cdot gridsize), y)$ in our 2D texture. Doing this comes with some overhead, we have to do additional calculations to find the new 2D position. Granted, this does not take very much time. The largest drawback is that we have to do trilinear filtering manually when rendering, instead of leveraging the hardware, which increases execution time. Therefore, if we had not limited ourselves to WebGL we could have skipped some intermediary steps in our implementation, thus making the algorithm faster.

Another limitation of WebGL is the lack of compute shaders. This would have been especially beneficial in the propagation step as it would simplify the implementation as we only need to propagate per cell. The simplification is due to not having to split up what would otherwise be a single compute shader into a vertex and fragment shader which require some setup before getting things to work without redundant calculations in the case of the propagation step. A more simple and straightforward implementation is the not the only benefit of using a compute shader in favor of traditional vertex and fragment shaders. A second benefit and perhaps of even more significance is that compute shaders allow for more effective parallel programming methods which may increase the performance of the program [38].

8.4 Comparative

Two of the techniques, 1 and 3, were significantly easier to implement than technique 2. This is partially because of the inherent complexity of the different techniques but also largely depends on how widely adopted the techniques are in industry and the availability of reference code and implementations. For technique 1 and 2 no resources other than the original papers were available. In the papers for technique 1 additional shader code was included in the appendix, while the paper for technique 2 did not describe all aspects of the implementation process in detail. In addition to the papers for technique 3 there are publicly available implementations to gather inspiration from.

Additionally, technique 2 involves long precomputation times which significantly slows down the time of iteration and debugging. Technique 1 and 3 do not require any precomputations before loading a scene, thus accelerating the time between iterations.

Technique 1 and 2 achieve better visual fidelity than technique 3 but this comes at the cost of performance and precomputation respectively. There were also differences between the two scenes evaluated. Technique 1 achieved the most ac-

curate results for the Sponza scene, and technique 2 achieved the most accurate results for the Living room scene.

In technique 1, a grid of probes which gives full visibility of the scene is needed. For the scenes we used this worked well but for very large and complex scene, an unreasonable amount of probes would be needed for full visibility. This limitation makes technique 1 much more suitable in certain contexts than others. For example, a game in which rooms are loaded when you enter would work well but large open world games would not. The other two techniques does not have this limitation, and technique 3 specifically is not very affected by its context.

While these three techniques are viable for real-time GI, modern technology such as the newly introduced DirectX Ray-Tracing [39] might fundamentally change the global illumination landscape. There is still a notable difference between path traced references and the explored techniques. With the increase of computational power new techniques which are only viable in the offline context might also be applicable in real-time.

9 Conclusion

All three techniques fulfill the criteria set up in section 1.1, and some of them excelled.

Compared to the ground truth images we consider all of the techniques to provide visually convincing and reasonably realistic global illumination, according to criterion 1, while also different.

We initially thought that handling dynamic light and objects would be a major issue for technique 1 and 2 due to the precomputations performed, but it turned out to work quite well in both.

There are also differences in performance between the techniques, but all techniques clearly achieved well above what is needed for smooth interactivity while also achieving adequate visual fidelity.

Even though there were differences in the difficulty of implementation, as discussed in 8.4, there were no technical constraints that we were unable to overcome, and all the techniques were implementable in a web-browser as shown in the results section.

We cannot conclusively decide on a single technique that is optimal in all regards, but we consider all of the three evaluated techniques to be viable for achieving real-time global illumination in web-browsers with respect to the limitations of such a platform.

References

[1] Can I use..., "Can I use? – WebGL 2.0," accessed on: Feb 07 2018. [Online]. Available: https://caniuse.com/#search=webgl2

[2] The Khronos Group Inc., "WebGL 2.0 Specification," 2017. [Online]. Available: https://www.khronos.org/registry/webgl/specs/2.0/

[3] T. Sherif, "PicoGL.js," accessed on: Mar 23 2018. [Online]. Available: https://tsherif.github.io/picogl.js/

[4] M. McGuire, *The Graphics Codex*, 2nd ed. Casual Effects, 2016. [Online]. Available: http://graphicscodex.com

[5] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering, Third Edition*. CRC Press, 2008.

[6] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 143–150. [Online]. Available: http://doi.acm.org/10.1145/15922.15902

[7] B. F. Janzen and R. J. Teather, "Is 60 fps better than 30?: The impact of frame rate and latency on moving target selection," in *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '14. New York, NY, USA: ACM, 2014, pp. 1477–1482. [Online]. Available: http://doi.acm.org/10.1145/2559206.2581214

[8] C. Ware and R. Balakrishnan, "Reaching for objects in VR displays: Lag and frame rate," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 4, pp. 331–356, Dec. 1994. [Online]. Available: http://doi.acm.org/10.1145/198425.198426

[9] M. McGuire, "Computer graphics archive," July 2017, `https://casual-effects.com/data`. [Online]. Available: https://casual-effects.com/data

[10] J. R. B. Edward H. Adelson, "The plenoptic function and the elements of early vision," in *Computational Models of Visual Processing*, M. Landy and J. A. Movshon, Eds. Cambridge, MA, USA: MIT Press, 1991, pp. 3–20. [Online]. Available: http://www1.cs.columbia.edu/~changyin/candidacy/AdelsonCMVP1991.pdf

[11] P.-P. Sloan, "Stupid spherical harmonics (sh) tricks," 2008. [Online]. Available: http://www.ppsloan.org/publications/StupidSH36.pdf

[12] Íñigo Quílez, "Spherical harmonics," May 2014, accessed on: May 14 2018. [Online]. Available: https://www.shadertoy.com/view/lsfXWH

[13] The Khronos Group Inc., "ARB_draw_buffers," 2008. [Online]. Available: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_draw_buffers.txt

[14] L. Williams, "Casting curved shadows on curved surfaces," in *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '78. New York, NY, USA: ACM, 1978, pp. 270–274. [Online]. Available: http://doi.acm.org/10.1145/800248.807402

[15] C. Dachsbacher and M. Stamminger, "Reflective shadow maps," 2005. [Online]. Available: http://www.klayge.org/material/3_12/GI/rsm.pdf

[16] W. Donnelly and A. Lauritzen, "Variance shadow maps," in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ser. I3D '06. New York, NY, USA: ACM, 2006, pp. 161–165. [Online]. Available: http://doi.acm.org/10.1145/1111411.1111440

[17] M. Thomas, "Realtime global illumination techniques collection," 05 2014, accessed on: May 14 2018. [Online]. Available: https://extremeistan.wordpress.com/2014/05/11/realtime-global-illumination-techniques-collection/

[18] M. McGuire, M. Mara, D. Nowrouzezahrai, and D. Luebke, "Real-time global illumination using precomputed light field probes," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '17. New York, NY, USA: ACM, 2017, pp. 2:1–2:11. [Online]. Available: http://doi.acm.org/10.1145/3023368.3023378

[19] A. Silvennoinen and J. Lehtinen, "Real-time global illumination by precomputed local reconstruction from sparse radiance probes," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 36, no. 6, pp. 230:1–230:13, Nov. 2017. [Online]. Available: https://doi.org/10.1145/3130800.3130852

[20] A. Kaplanyan, "Light propagation volumes in CryEngine 3," 2009. [Online]. Available: http://www.crytek.com/download/Light_Propagation_Volumes.pdf

[21] Epic Games, "Light Propagation Volumes," accessed on: May 12 2018. [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/LightPropagationVolumes

[22] Z. H. Cigolle, S. Donow, D. Evangelakos, M. Mara, M. McGuire, and Q. Meyer, "A survey of efficient representations for independent unit vectors," *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 2, pp. 1–30, April 2014. [Online]. Available: http://jcgt.org/published/0003/02/01/

[23] G. J. Ward, F. M. Rubinstein, and R. D. Clear, "A ray tracing solution for diffuse interreflection," *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 85–92, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/378456.378490

[24] J. Krivanek, P. Gautron, S. Pattanaik, and K. Bouatouch, "Radiance caching for efficient global illumination computation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 5, pp. 550–561, Sept 2005. [Online]. Available: https://ieeexplore.ieee.org/document/1471692/

[25] I. Castaño, "thekla_atlas," accessed on: May 13 2018. [Online]. Available: https://github.com/Thekla/thekla_atlas

[26] D. Dunbar and G. Humphreys, "A spatial data structure for fast poisson-disk sample generation," in *ACM SIGGRAPH 2006 Papers*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006, pp. 503–508. [Online]. Available: http://doi.acm.org/10.1145/1179352.1141915

[27] P.-P. Sloan, J. Hall, J. Hart, and J. Snyder, "Clustered principal components for precomputed radiance transfer," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 382–391, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882281

[28] J. W. Demmel, *Applied numerical linear algebra*. Siam, 1997, vol. 56.

[29] V. Rokhlin, A. Szlam, and M. Tygert, "A randomized algorithm for principal component analysis," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, pp. 1100–1124, 2009. [Online]. Available: https://arxiv.org/abs/0809.2274v4

[30] K. Balasubramanian, K. Yu, and G. Lebanon, "Smooth Sparse Coding via Marginal Regression for Learning Sparse Representations," *ArXiv e-prints*, Oct. 2012. [Online]. Available: https://arxiv.org/abs/1210.1121v1

[31] I. Ramírez, F. Lecumberry, and G. Sapiro, "Sparse modeling with universal priors and learned incoherent dictionaries," University of Minnesota. Institute for Mathematics and Its Applications, 9 2009. [Online]. Available: http://hdl.handle.net/11299/180327

[32] C. R. Genovese, J. Jin, L. Wasserman, and Z. Yao, "A comparison of the lasso and marginal regression," *Journal of Machine Learning Research*, vol. 13, no. Jun, pp. 2107–2143, 2012. [Online]. Available: http://www.jmlr.org/papers/v13/genovese12b.html

[33] R. Ramamoorthi and P. Hanrahan, "An efficient representation for irradiance environment maps," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 497–500. [Online]. Available: http://doi.acm.org/10.1145/383259.383317

[34] A. Kaplanyan and C. Dachsbacher, "Cascaded light propagation volumes for real-time indirect illumination," 2010. [Online]. Available: http://www.crytek.com/download/20100301_lpv.pdf

[35] S. H. Ahn, "Opengl projection matrix," 2008. [Online]. Available: http://www.songho.ca/opengl/gl_projectionmatrix.html

[36] A. Kirsch, "Light propagation volumes – annotations," 2010. [Online]. Available: http://data.blog.blackhc.net/2010/07/lpv-annotations.pdf

[37] ——, "Light propagation volumes – corrections," 2010. [Online]. Available: http://data.blog.blackhc.net/2010/07/lpv-corrections.pdf

[38] F. Sans and R. Carmona, "A comparison between gpu-based volume ray casting implementations: Fragment shader, compute shader, opencl, and cuda," *CLEI Electronic Journal*, vol. 20, no. 2, pp. 7:1–7:19, 2017. [Online]. Available: http://www.clei.org/cleiej-beta/index.php/cleiej/article/view/26

[39] M. Sandy, "Announcing microsoft directx raytracing!" March 2018, accessed on: May 14 2018. [Online]. Available: https://blogs.msdn.microsoft.com/directx/2018/03/19/announcing-microsoft-directx-raytracing/