HASSO-PLATTNER-INSTITUT
Fachgebiet Computergrafische Systeme

Design and Implementation of a Many-Light Method for Real-Time Dynamic Global Illumination

Masterarbeit
zur Erlangung des akademischen Grades
"Master of Science"
(M.Sc.)
im Studiengang IT-Systems Engineering
des Hasso-Plattner-Instituts an der
Universität Potsdam

vorgelegt von

Johannes Linke

Aufgabenstellung und Anleitung:
Prof. Dr. Jürgen Döllner
Daniel Limberger

Potsdam,
23. Dezember 2016

Contents

Abstract

Global illumination is one of the most thoroughly investigated fields in computer graphics. Its applications are numerous, from offline rendering to real-time applications, for photorealistic and non-photorealistic rendering, and on static or dynamic geometry, each coming with their own set of requirements in terms of flexibility, performance, and quality. A multitude of widely differing approaches have been invented to simulate global illumination for each of the different sets of requirements. One of these approaches are many-light methods. They place lots of *virtual point lights* in illuminated areas of a scene. These virtual point lights are themselves used to light the scene, thereby simulating indirect lighting. The concept is highly scalable and is used in both offline and real-time applications.

One of the most challenging application areas of global illumination are real-time applications with dynamic geometry. While moving geometry prevents most kinds of precomputation, having only a few milliseconds of computation time forces researchers and application developers to make compromises on the quality or constrain their system to a very specific use case. As a result, all available global illumination algorithms have significant drawbacks, for example not achieving real-time performance, not fully supporting dynamic geometry, supporting only certain kinds of scenes, producing low-quality output, or not being scalable to low-end devices and high quality levels.

In this thesis, we explore the field of real-time global illumination systems by implementing and optimizing a many-light algorithm. The focus lies on improving imperfect shadow maps, which are approximate shadow maps that can efficiently be rendered for hundreds or thousands of lights. Additionally, we apply clustered deferred shading, an optimization that performs light culling, to many-light methods, and present an efficient implementation of interleaved shading, another optimization technique. Although coming with its own set of drawbacks, the presented rendering system runs in real-time on current commodity hardware and requires no precomputation. The results show that the achieved performance is more than sufficient to be used in real-time applications. However, the output quality leaves much to be desired, primarily because of several flaws of imperfect shadow maps, and requires a more fundamental change of approach to visibility testing.

Zusammenfassung

Globale Beleuchtung ist eines der am meisten erforschten Gebiete der Computergrafik. Es gibt eine Vielzahl von Anwendungsmöglichkeiten, vom Offline-Rendering zu Echtzeitanwendungen, für fotorealistische und nicht-fotorealistische Bildsynthese, und für statische und dynamische Geometrie. Jede der Anwendungen hat dabei verschiedene Anforderungen in Bezug auf Flexibilität, Performance, und Qualität. Die Forschung hat eine große Menge von sich teilweise stark unterscheidenden Ansätzen für die Simulation globaler Beleuchtung unter Berücksichtigung der verschiedenen Anforderungen hervorgebracht. Einer dieser Ansätze sind die Many-Light-Methoden. Bei diesem Ansatz wird eine große Menge von virtuellen Punktlichtquellen in beleuchteten Bereichen der Szene erstellt. Diese Lichtquellen beleuchten ihrerseits die Szene und simulieren somit reflektiertes Licht. Das Konzept ist in hohem Maße skalierbar und wird sowohl im Offline-Rendering als auch in Echtzeitanwendungen verwendet.

Eine der herausfordernsten Anwendungsbereiche für globale Beleuchtung sind Echtzeitanwendungen mit dynamischer Geometrie. Letzteres verhindert den Einsatz von aufwendigen Vorberechnungen, und da nur wenige Millisekunden zur Berechnung zur Verfügung stehen, müssen Forscher und Anwendungsentwickler Kompromisse bei der Qualität eingehen oder ihr Konzept auf einen spezifischen Anwendungsfall beschränken. Dadurch haben alle bestehenden Algorithmen zur Berechnung globaler Beleuchtung deutliche Nachteile, z. B. funktionieren sie nicht in Echtzeit, unterstützen keine dynamische Geometrie oder nur bestimmte Arten von Szenen, haben im Ergebnis eine geringe Qualität oder skalieren nicht von Low-End-Geräten bis zu hohen Qualitätsanforderungen.

In dieser Arbeit untersuchen wir das Feld der globalen Beleuchtung in Echtzeit, indem wir einen Many-Light-Algorithmus implementieren und optimieren. Der Fokus liegt dabei auf der Verbesserung von *Imperfect Shadow Maps*, d. h. kleinen, approximierten Shadow Maps, die effizient zu hunderten generiert werden können. Zusätzlich wenden wir *Clustered Deferred Shading* an, eine Optimierung zum Culling von Lichtquellen, und präsentieren eine effiziente Implementierung von *Interleaved Shading*, einer weiteren Optimierungstechnik. Auch wenn das vorgestellte Renderingsystem seine eigenen Nachteile hat, läuft es auf handelsüblicher Hardware in Echtzeit und benötigt keine Vorberechnungen. Die Ergebnisse zeigen, dass die erreichte Performance mehr als genug für Echtzeitanwendungen ist; allerdings genügt die Ausgabe nicht höheren Qualitätsanforderungen. Verantwortlich sind vor Allem mehrere Schwächen der *Imperfect Shadow Maps*, die sich vermutlich nur durch einen fundamental anderen Ansatz zum Testen der Sichtbarkeit lösen lassen.

Chapter 1

Introduction

Correctly simulating the behavior of light is one of the largest research fields in the realm of computer graphics. In fact, since the output of any image synthesis algorithm is essentially the amount of light reaching a virtual camera, one could say that almost all techniques developed for synthesizing images are actually means to simulate light, either more correctly or using less hardware resources.

The complexity of simulating light arises from the multitude of different paths that individual photons can take from a light source to the camera. Along the way, photons can be reflected and refracted an arbitrary number of times, or can even be absorbed and not reach the camera at all. As a way to reduce the computational complexity of simulating these processes in a physically correct manner, computer graphics researchers have separated the interaction of light and matter into several visually distinct phenomena, which are then simulated separately with wildly differing approaches.

Some of these phenomena are direct and indirect lighting and shadowing, large-scale and small-scale lighting, diffuse and specular reflections, subsurface scattering, caustics, different types of lights such as point, directional, and area lights or light emitting surfaces, participating media like fog or smoke, and transparent surfaces. Each of these is of considerable complexity, has been studied extensively and most are still actively researched.

Of particular importance for realistic light simulation is large-scale and indirect lighting, which is often called “global illumination”. For instance, in indoor scenes where the sun shining through windows is the only light source, most parts of the scene receive light only indirectly and after multiple reflections. Without global illumination, those areas would receive no light at all and appear completely black. Thus, physically correct or at least plausible global illumination is of high importance for the film industry, video game creators, architectural and e-commerce visualizations, and many more.

Unfortunately, large-scale and indirect lighting is inherently expensive to compute since the light needs to be transported over long distances, in arbitrary directions, and with multiple reflections. In offline rendering systems as they are used by, for example, the film industry, the available computation time and power makes this less of a problem than it is for real-time applications. Here, compromises have been made, such as long precomputation times, disallowing geometry and/or light movement, and only simulating diffuse and not specular reflections, to mimic at least some of the effects of global illumination.

Being both important for high-quality renderings and expensive to compute, global illumination has become a particularly large research area and has generated a lot of different approaches for simulating or approximating it, each with different application areas, advantages, and disadvantages. One of these approaches is instant radiosity (Keller, 1997), also called many-light methods, a well-known family of algorithms that scale from real-time applications to offline rendering. They use a rather intuitive model to approximate lighting and they can be used to simulate other lighting effects besides global illumination as well.

In this thesis, we present an implementation of a rendering system using the many-light approach to simulate global illumination. The system requires no precomputation and needs only several milliseconds for its computations, making it suitable for real-time applications. For indirect shadowing, the software renders several hundred imperfect shadow maps (Ritschel *et al.*, 2008) each frame. Our main contributions are:

- the application of a high-quality postprocessing (Marroquim *et al.*, 2007) for higher-quality imperfect shadow maps
- an efficient algorithm for interleaved sampling (Keller *et al.*, 2001) using compute shaders[1]
- the application of clustered deferred shading (Olsson *et al.*, 2012) as a performance optimization to many-light methods.

While lights and geometry are allowed to move, the system does not provide temporal stability in this case. Furthermore, it simulates only one light bounce, i. e., light is reflected only once before illuminating surfaces visible to the camera. However, the approach is not inherently limited in this regard and can be extended to multiple bounces in future work.

The full source code of the implementation is available in the project's repository[2] under the MIT license.

This thesis is structured as follows: The next chapter details the global illumination effect and related work on it, before focusing on many-light methods by explaining the general idea and showing different approaches that have been investigated in the past. Chapter 3 describes, on a conceptual level, the implemented system with its different components. The implementation with its data layout and performance considerations is detailed in Chapter 4, and Chapter 5 presents performance measurements, quality assessments, and discussions on the chosen approaches. Concluding remarks are provided in Chapter 6.

[1]https://www.opengl.org/wiki/Compute_Shader
[2]https://github.com/karyon/many-lights-gi

Chapter 2

Introduction to Global Illumination

This chapter starts by providing a brief overview of lighting techniques in computer graphics and locate and distinguish global illumination in this vast research field. Subsequently, the foundation for the remainder of this work is laid by introducing the conceptual and theoretical foundations of global illumination in general and many-light methods in particular, combined with an overview of the related work on those topics.

2.1 A High-Level Overview of Lighting in Computer Graphics

Before examining the different lighting effects simulated in computer graphics, it is worth pointing out that in real-world physics, most of the distinctions made later in this section do not exist. Many effects and phenomena in lighting can be reduced to a basic set of interactions of photons, which are emitted by light sources and then reflected, refracted, or absorbed by matter.

In practice, simulating individual photons and atoms is obviously infeasible and unnecessary for most computer graphics applications. Instead, computer graphics researchers have examined the different real-world light phenomena and approximated them separately, using different approaches and approximations for each of them to stay within their respective performance budget.

In the following a brief and non-comprehensive overview of these lighting effects is provided and the term *global illumination* is defined more precisely. The most top-level categorization of lighting effects divides them into the interaction of light with scene surfaces, and the transport of light through empty space in the scene.

Light-Surface Interaction

To determine how much and in what direction incoming light is reflected from surfaces, *bi-directional reflectance distribution functions* (BRDFs) are used. While the recent shift to physically-based BRDFs has brought more unified models to real-time computer graphics, several materials such as transparent (e. g., glass) or highly translucent ones (e. g., skin) are still handled separately.

Light Transport

In order for light to interact with surfaces, it needs to be "transported" there from light sources. The simplest case, light that hits only one surface before reaching the camera, is called **direct light**. To determine surfaces that are directly lit by a certain light source, shadow maps are commonly used in real-time computer graphics. Direct light usually has a single point as origin in the case of point lights, or consists of entirely parallel light rays in the case of directional lights. This structuredness makes it relatively easy to compute, in terms of both complexity and performance.

Indirect light on the other hand has bounced off of at least two surfaces before reaching the camera. Since each point that is directly lit now reflects light into arbitrary directions, this process is much less structured and thus harder to compute. To manage this complexity, indirect light has been separated into small-scale and large-scale indirect light. An intermediate step for medium-scale indirect lighting has also been proposed (Reed, 2012).

The computation of **small-scale indirect light** has been made popular with *screen-space ambient occlusion* (SSAO, Mittring, 2007). While this should be more accurately described as small-scale indirect *occlusion*, since it only darkens occluded areas, *screen-space directional occlusion* (SSDO, Ritschel *et al.*, 2009b) actually performs a form of indirect lighting, albeit a limited one. Jiménez (2016) has finally put this technique onto a physically based foundation.

Small-scale indirect lighting techniques such as SSAO are important for accentuating small geometric details in objects and providing a sort of contact shadow as a visual cue for the proximity of objects to each other.

In contrast, **large scale indirect lighting** provides more realistic lighting for entire scenes. While it has been computed in real-time for static scenes for a while, large-scale indirect lighting for fully dynamic scenes has yet to reach widespread use in real-time applications. More on this issue follows in the next section. The term **global illumination** (GI) has been used for varying sets of lighting effects throughout the literature. In this thesis, it is used to describe the lastly mentioned aspect, i. e., large-scale indirect lighting.

2.2 Introduction to Global Illumination

This section introduces global illumination by outlining its advantages and describing its theoretical foundations. Thereafter it gives a brief overview of techniques used previously to simulate (non-dynamic) global illumination before detailing the more recent related work in this field, and structures the process of computing global illumination by identifying individual building blocks. For a comprehensive introduction to global illumination, see Ritschel *et al.* (2012).

2.2.1 Motivation

The single overarching advantage of simulating global illumination in graphics applications is the added realism which leads to a host of benefits. An obvious one is the greater immersion achieved in movies and video games, but also architectural and e-commerce visualization applications can provide better assistance to its users through more realistic renderings. For instance, in architectural visualization, the colors of the furniture, carpets, decoration etc. are influencing the overall tone of the scene, e. g., by coloring the walls slightly, leading to a better basis for real-world design decisions.

Real-time techniques that allow camera movements in a given scene and lighting setup on the one hand enable approximating global illumination in applications that are inherently real-time, e. g., video games. On the other hand, even applications that do not require real-time frame rates can benefit from such techniques. For instance, while the features that are covered in this thesis are basically solved for offline rendering, the film industry can benefit from real-time global illumination techniques during production by means of faster testing of, e. g., different camera angles. Archviz applications can serve more use cases by implementing interactive walkthroughs.

Allowing changing geometry provides additional benefits: Games can create new experiences with changing level geometry, e. g., through destruction; level designers and lighting artists can expect a more natural workflow without having to resort to workarounds like fake lights; artists in filmmaking benefit from shorter iteration cycles; and visualization tools provide immediate feedback to changes in, e. g., interior decoration in the case of archviz.

2.2.2 Theory

This section will explain the significance of global illumination using the rendering equation (Kajiya, 1986). The most basic form of the rendering equation is as follows:

$$L_o(x \quad _o) = L_e(x \quad _o) + L_r(x \quad _o) \tag{2.1}$$

where L_o describes the outgoing radiance from a surface point x in the direction $\quad_o$ as the emitted radiance L_e plus the reflected radiance L_r. The significance of this equation comes from the fact that the outgoing radiance of a point towards the camera determines its color in a rendering.

The reflected radiance can be defined as

$$L_r(x \quad _o) = \int_\Omega f_r(x \quad _i \quad _o) L_i(x \quad _i)(\quad _i \quad \boldsymbol{n}) d \quad _i \tag{2.2}$$

where Ω is the hemisphere located at point x and oriented towards the surface normal $\boldsymbol{n}$ at point x, f_r is the BRDF of the surface and L_i the incident radiance. While this is the most common representation of the rendering equation, Kajiya (1986) defined the reflected radiance using an integral over all surface points, not over all directions:

$$L_r(x \quad _o) = \int_S f_r(x \quad _i \quad _o) L_i(x \quad _i)(\quad _i \quad \boldsymbol{n}) V(x\, x \quad) dx \tag{2.3}$$

where $\ _i$ is redefined as the normalized difference vector between x and x and S is the (infinite) set of all surface points. Since not all surface points are visible from x, a visibility term V has been added that is 1 if x and x are mutually visible, 0 otherwise.

This equation illustrates a major difficulty when simulating global illumination, namely the integral that needs to be solved for the infinite set of surface points. To approximate this integral, often clusters of points are selected to form surface patches. Even then, the performance requirements are generally too high for real-time computer graphics. As a result, this integral is either solved in a preprocessing step for static parts of the scene, or solved only for the set of points that are considered light sources, discarding a large portion of the light energy in the scene and heavily biasing the rendered output. In fact, area lights are often ignored or approximated because they again would require solving this integral for an infinite number of points.

Another difficulty made explicit by Equation (2.3) is the visibility term. Even with an algorithm that transfers light between points or surface patches at a sufficient resolution and sufficient speed, the visibility term requires the addition of another algorithm that tests for each pair of surface elements whether any other surface element is located between them. To perform such visibility tests efficiently, acceleration structures are needed that allow to quickly traverse the scene's geometry. However, these data structures are often difficult to update with dynamic geometry and do not map well to GPUs. This often makes visibility testing one of the most computationally intensive parts of global illumination algorithms.

2.2.3 The Path to Real-Time Dynamic Global Illumination

In interactive 3D graphics applications, global illumination has been ignored for a long time. To avoid completely unlit areas, an ambient term has been used that uniformly adds light to all surfaces.

For large-scale indirect lighting, light maps have seen widespread use since they have been introduced in the video game Quake (Abrash, 1997). This technique is inherently static since the texturing of scene surfaces is an offline preprocessing step, and it is inherently large-scale since large texels are used to avoid excessive memory requirements.

Games have employed variations of *precomputed radiance transfer* (PRT, Sloan *et al.*, 2002) to accommodate for changing lighting conditions. For instance, Stefanov (2012) uses probes instead of the scene geometry to store the PRT results to be able to light dynamic objects. However, even though dynamic objects receive indirect lighting, they are unable to affect indirect lighting of other objects, a commonly made compromise.

Besides not fully supporting dynamic objects, the long precomputation times of conventional global illumination methods are a common hindrance in the workflow of artists. Global illumination techniques that support dynamic objects and require no expensive precomputation have yet to reach widespread use in real-time applications due to the high performance requirements.

2.2.4 Components of Real-Time Dynamic Global Illumination

In order to better understand the next section discussing the current research around real-time dynamic global illumination, we have identified three stages that most real-time global illumination methods consist of:

Direct lighting or light injection. Since light sources can be very (if not infinitely) small and can be difficult to process for some techniques, the first step of the light propagation, sometimes called "light injection", can be separated and used to compute areas that are directly lit by the scene lights. These areas are then used as starting points for the global illumination algorithm. Other techniques use this approach purely as an optimization, since the first light bounce can be easier to compute than subsequent bounces.

Light propagation. Starting from the light source or directly lit surfaces, this step sends out the light and records where it hits scene geometry. This might be done repeatedly to simulate multiple light bounces.

Final gathering. Several techniques perform one last step that, instead of propagating light from lit surfaces into all directions, gathers light from lit surfaces into areas that are visible to the camera. This is a common optimization that guarantees that the calculations are relevant to the currently rendered frame, whereas the light propagation step is often more unguided and oblivious to the current viewport.

Note that not all techniques implement all of these steps. For instance, performing direct lighting and a final gathering step while omitting the light propagation step is sufficient for simulating one indirect bounce. Brute-force ray tracing can be done by simulating only the light propagation step, and light propagation volumes (Section 2.3.2) omit the final gathering step.

There are two more essential design decisions to be made:

Visibility testing. To prevent light from incorrectly shining through objects or walls, the visibility term in Equation (2.3) needs to be solved. Since brute-force ray casting is prohibitively slow, it requires special acceleration structures or alternative approximations, like replacing ray casts with (a moderate amount of) shadow maps. As mentioned previously, this component uses a substantial if not most of the computation time for many of the global illumination approaches presented in the next section.

Choice of receiving elements. The final gathering step collects light into areas that are visible to the camera. These areas can be represented by several forms of *receiving elements*. Besides directly using pixels in screen space, other choices are texels in texture space, surfels or voxels in world or view space, or other virtual objects placed in the scene, e. g., in the form of spherical harmonics, which are then interpolated for shading individual pixels. Technically, the light propagation step also necessarily stores its results in some form of receiving elements, which might differ from the ones that are used during final gathering. In this thesis however, the term describes those data structures that are used to perform the actual per-pixel shading.

As with most lighting concepts, real-time global illumination is often separated into diffuse and specular components, since the diffuse component is both easier to simulate with plausible results due to its low-frequency nature, and more important for the visual quality compared to the specular component. For the same reasons, only the diffuse component is covered in this thesis.

2.3 Previous Work on Real-Time Dynamic Global Illumination

This section presents previous work on global illumination while focusing on methods that work in real-time and support dynamic scenes, excluding many-light methods which are covered separately in the next section. A more comprehensive study of interactive global illumination methods is provided by Ritschel *et al.* (2012).

2.3.1 Point-Based Approaches

Based on the work of Bunnell (2005), point-based global illumination has been used for offline rendering in the film industry (Christensen, 2008). Despite its name, the geometry representation used to approximate the scene geometry consists of disk-shaped surface elements (*surfels*). These surfels are then organized into the leaves of a tree structure, while the higher-level nodes are aggregate representations of all nodes they contain. This tree is then used to compute visibility and propagate light between the surfels.

While Bunnell (2005) propose a rough approximation for computing visibility, Christensen (2008) employs ray tracing or spherical harmonics based on the distance between surfels to compute fairly accurate approximations of global illumination. Ritschel *et al.* (2009a) achieve interactive frame rates by utilizing the GPU to perform the final gathering step, which renders the scene into a *micro-buffer* for each receiving element (in their case, screen-space pixels). Due to the large amounts of surfels required and because the tree structure can be expensive to update when geometry moves, this technique is best suited for static or very small dynamic scenes.

2.3.2 Light Propagation Volumes

Initially proposed by Kaplanyan (2009) and extended by Kaplanyan *et al.* (2010), this technique reduces the scene's geometry to two voxel grids, called *light propagation volume* and *geometry volume* respectively. Thereafter, light is injected from primary light sources into the light propagation volume, and the scene geometry is inserted into the geometry volume. With this, the light is iteratively propagated from each illuminated voxel to its neighboring voxels until the geometry volume indicates that the path is occluded. However, since inaccuracies of the propagation process accumulate over the iterations, this approach is inaccurate when dealing with long distances between sending an receiving surfaces.

2.3.3 Voxel Cone Tracing

Similar to light propagation volumes, the scene is first reduced to voxels and light from primary sources is injected. However, instead of propagating the light through the grid, the light is collected starting at receiving elements, usually pixels in screen space, by tracing cones through the grid. While the original proposal (Crassin *et al.*, 2012) uses a sparse voxel octree to represent the scene, Panteleev (2015) introduces clip-maps that use several levels of equally-sized voxel grids to represent the scene with varying resolution depending on the distance to the camera. Both approaches can compute specular reflections for moderately glossy surfaces. While the voxel octree is expensive to update with dynamic objects, the clip-maps are faster in this regard, but still need relatively large amounts of VRAM for higher quality levels. Lower quality levels result in noticeable voxelization artifacts.

2.3.4 Ray Tracing

Several attempts have been made to adapt ray tracing to be usable in real-time contexts. Thiedemann *et al.* (2011) trace rays through a voxel grid, but have to limit the ray's maximum distance for performance reasons and, similar to voxel cone tracing, suffer from high memory requirements. Tokuyoshi *et al.* (2012) trace rays using rasterization, but at the cost of a severe performance impact since they render the whole scene multiple times. Chen *et al.* (2016) use one bit per voxel to indicate whether it is opaque or not, and look up lighting information directly in an RSM during the final gather phase. They too achieve merely interactive frame rates.

2.3.5 Radiance Caching

Partly orthogonal to the previous sections, radiance caches are a different form of receiving elements that are placed in world or screen space and capture the incoming radiance. During actual shading, the nearest caches are interpolated. The major advantage is the reduction of the number of receiving elements for which incoming light needs to be gathered. The original proposal cached irradiance values (Ward *et al.*, 1988). The resulting loss of detail during interpolation and inability to compute specular reflections was overcome by radiance caching (Krivanek *et al.*, 2005) at a performance loss. Scherzer *et al.* (2012) achieved real-time frame rates for static geometry and another improvement in efficiency was proposed by Rehfeld *et al.* (2014). Radiance caching is orthogonal to the other techniques in so far as visibility still needs to be solved by other means, and using it introduces new problems such as temporal stability of the cache placement.

2.3.6 Screen-Space Approaches

Screen-space approaches inherently suffer from the lack of information about objects that are outside of the view frustum or occluded. The latter is alleviated by deep G-buffers (Mara *et al.*, 2014; Mara *et al.*, 2016), but working only within the frustum, even this approach is limited to small to medium-scale indirect illumination.

2.4 Introduction to Many-Light Methods

This section will introduce many-light methods by first explaining the idea behind the technique and the effects that can be simulated, and then identifying the different components of a many-light rendering system.

2.4.1 Overview

Many-light methods as another means to compute global illumination originate in instant radiosity (Keller, 1997). In the original paper, photons are traced through the scene, similar to photon mapping (Jensen, 1996). With each bounce, however, instead of storing the photon in a photon map, instant radiosity creates a new *virtual point light* (VPL). These point lights illuminate the scene and thereby simulate light reflections (Figure 2.1). While the concrete method of creating VPLs often differs, many-light methods share the idea of approximating various lighting effects through large numbers of VPLs.

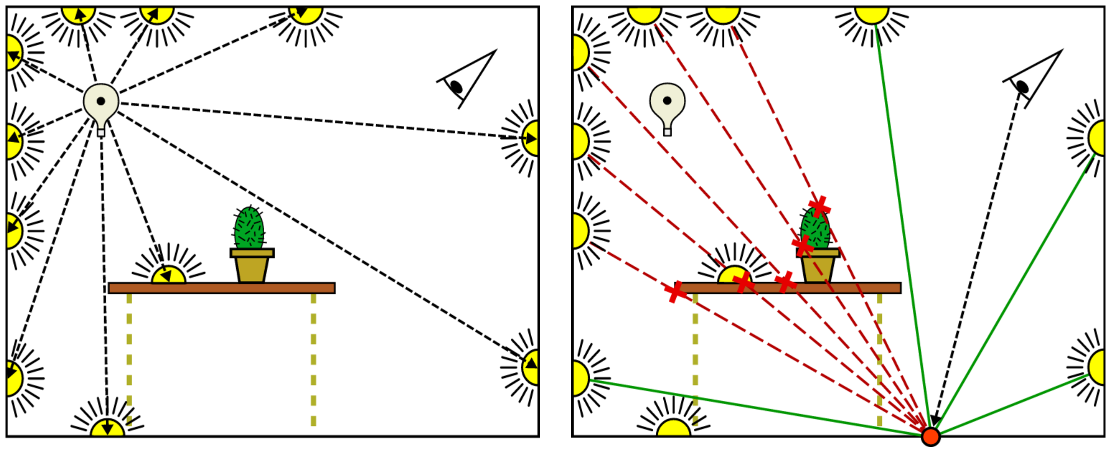

Figure 2.1: *Illustration of the many-light concept. Left: VPLs are created on surfaces that are directly lit by a light source. Right: When shading a surface point (orange), all VPLs that are visible for that point are used to indirectly light it. Figures reprinted from Laine et al. (2007).*

The use of many-light methods is intriguing since they are a natural and intuitive model of real-world light interactions. In addition the concept is highly scalable: Many-light methods have been used for real-time applications with up to a few thousand lights to offline rendering with millions of lights. Given a sufficient number of lights, the concept is capable of accurately simulating advanced effects like subsurface scattering and participating media. Dachsbacher *et al.* (2014) gives an overview of the higher-end spectrum of many-light methods.

When applied to real-time applications, the available performance budget enforces the use of a limited number of lights. With a few thousand lights at most, many effects are not possible to reasonably approximate anymore. However, the low-frequency nature of diffuse reflections enables several approximations and optimizations that make a convincing global illumination effect feasible to create. Using more or less conventional point light sources has another advantage, at least in the context of real-time applications: Many techniques from classical real-time rendering are applicable, such as point light rendering including conventional shadow maps for visibility testing. This allows for simple implementations of the basic concepts, even though more advanced techniques are necessary to achieve high performance and quality levels.

2.4.2 Components of Many-Light Methods

This section will take the components of global illumination identified in Section 2.2.4 and explain how they translate to many-light methods. Two new components are added that are unique to many-light methods, namely VPL placement and mitigating singularities.

Direct lighting or light injection. In theory, many-light methods do not need a light injection step since they are usually iterative processes that start with the scene lights and from there on, create new lights wherever the current light set illuminates the scene. In practice, the first bounce is often handled separately for performance reasons, and because the VPLs have different characteristics than most scene lights.

Light propagation. As just mentioned, many-light methods can simulate an arbitrary number of light bounces by repeating the propagation step with an expanding set of VPLs. However, many techniques (and this thesis) simulate just the first bounce, since that alone provides a large quality enhancement over simulating no global illumination at all, and subsequent bounces are more complex in terms of implementation and computation.

VPL placement. Since the budget of VPLs is limited, it is desirable to get the maximum effect out of each VPL. Thus, much research has gone into placing VPLs where they contribute the most to the output image, while at the same time not introducing any bias. Both the light injection and light propagation step create VPLs and can take considerable amounts of computation time through complex sampling methods. Some techniques also do not distinguish between light injection and light propagation and have a unified process for placing VPLs.

Final gathering. Most many-light methods perform final gathering per screen-space pixel. There are two main approaches commonly referred to as splatting and gathering. Both are too slow to use every light to shade every pixel, therefore optimizations and approximations are used. More on this in the next section.

Visibility testing. Many-light methods provide no means of visibility computation; these need to be solved separately. The advantage of many-light methods is that each receiving element only needs to test the (bounded) number of VPLs for visibility, not an arbitrarily high number of scene elements. In contrast to other global illumination methods, this makes it possible (although not performant) to use shadow maps for visibility testing.

Choice of receiving elements. As mentioned before, screen-space pixels are most commonly used, albeit often downscaled or combined with interleaved sampling (Section 2.5.2). The other options listed in Section 2.2.4 are technically possible as well, but haven't been combined with many-light methods yet.

Mitigating singularities. Since the entire energy of the indirect light bounces is concentrated into a few infinitely small light sources, the areas near these light sources receive too much light. This effect is distinctly visible as bright spots around the VPL's location (Figure 2.2), making it necessary for many-light methods to alleviate this artifact.

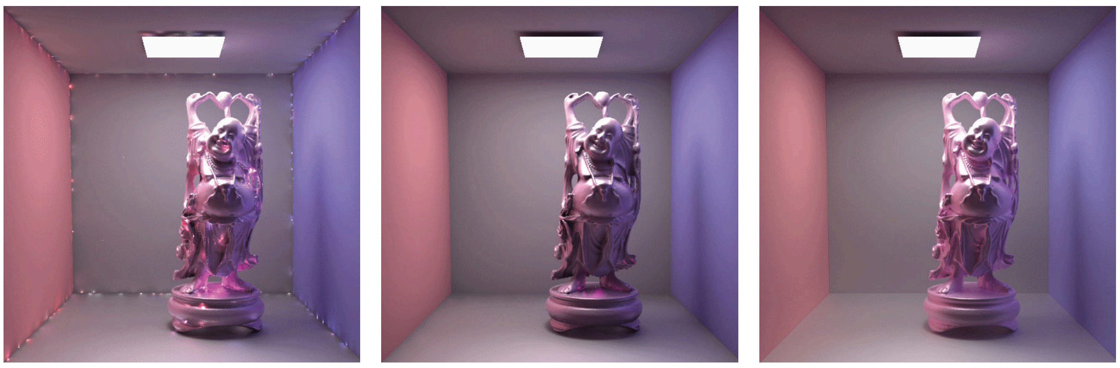

Figure 2.2: *A rendering generated using a many-light method. (a) Naive approaches produce singularities near the VPL's location. (b) Clamping the light attenuation term makes the problem less apparent, but causes bias in the form of darkening in the corners and on the statue. (c) An unbiased rendering for comparison. Figure reprinted from Dachsbacher et al. (2014).*

2.5 Previous Work on Real-Time Many-Light Methods

This section will present several many-light methods while concentrating on those that are applicable to real-time rendering. It is structured roughly according to the different design decisions to be taken when applying many-light methods: VPL sampling, computing visibility, final gathering, and mitigating singularities. The light transport step is not covered separately as it is often either omitted, simulating only one bounce, or it is an integral part of the VPL sampling algorithm. The choice of receiving elements is not covered as well, as most papers in this area simply use screen-space pixels. Dachsbacher *et al.* (2014) provide another overview of many-light methods, including those unsuitable for real-time rendering.

2.5.1 Virtual Point Light Sampling

While Keller (1997) proposes an approach similar to photon mapping to create VPLs, real-time applications are in need of something more performant. A commonly used technique are *reflective shadow maps* (RSMs, Dachsbacher *et al.*, 2005), which use rasterization to create first-order VPLs. While in this paper, each pixel of the RSM is considered a VPL and during gathering, a random subset of all VPLs is sampled, beginning with Dachsbacher *et al.* (2006) most papers sample the RSM to create a fixed set of VPLs which is used during shading.

Georgiev *et al.* (2010) and Ritschel *et al.* (2011) sample the RSM to create a set of VPLs with (estimated) high contributions to the final image. Dong *et al.* (2009) and Prutkin *et al.* (2012) cluster several samples to form *virtual area lights*. They observe that far fewer virtual area lights than VPLs are necessary to achieve the same quality at a minor performance expense.

Most of these approaches suffer from poor temporal stability. To improve on that, Laine *et al.* (2007) update only a portion of the VPLs per frame but introduce latency to the indirect light, additionally dynamic objects can receive but not bounce light. Barák *et al.* (2013) provide temporally stable results with dynamic scene geometry, but not with moving light sources. Hedman *et al.* (2016) achieve near-optimal temporal stability even with moving light sources while maintaining high per-image accuracy at real-time frame rates. They use one classic shadow map per VPL though, which must be updated lazily to stay within real-time limits.

2.5.2 Final Gathering

Splatting techniques have been used to add a light's contribution to the rendered image (Dachsbacher *et al.*, 2006; Nichols *et al.*, 2009b), but do not utilize modern GPUs efficiently. Instead, gathering approaches are commonly used today, employing interleaved sampling (Keller *et al.*, 2001) to reduce the number of lights processed per pixel, in combination with an edge-aware blur similar to Laine *et al.* (2007). Segovia *et al.* (2006b) improve the cache efficiency of interleaved sampling.

2.5.3 Visibility Computation

There are several approaches to calculate visibility between VPLs and the scene parts visible to the camera. Classic shadow maps are a fairly exact solution, but cannot be updated every frame for the several hundreds or even thousands of VPLs.

A popular approach are *imperfect shadow maps* (ISMs, Ritschel *et al.*, 2008), which use a precomputed set of points as scene representation to quickly render large amounts of small and inaccurate shadow maps. Ritschel *et al.* (2011) extend the approach to fully dynamic scenes among other improvements. Barák *et al.* (2013) use tessellation to compute the point set, eliminating the need to keep a separate point set updated, making it inherently dynamic, and providing better performance for larger point sets.

Ray tracing has been proposed to compute visibility as well (e. g., Segovia *et al.*, 2006a), but suffers from the usual drawbacks of ray tracing in a real-time context. A voxel-based scene representation has also been used to perform visibility queries for many-light techniques (Sun *et al.*, 2015).

2.5.4 Mitigating Singularities

In naive implementations of many-light techniques, bright spots will appear near the VPL's positions due to the light's attenuation term approaching infinity. A common approach is to clamp the term (Figure 2.2). This introduces bias, which can be compensated, e. g., in screen space (Novák *et al.*, 2011). Singularities can also be avoided through more advanced light representations (Tokuyoshi, 2015).

Chapter 3

Concept

In this chapter, the presented global illumination system is explained on a conceptual level. After providing an overview of the rendering pipeline in the next section, the chosen approaches for the different components required for global illumination are outlined. Implementation details are deferred to the next chapter.

3.1 Global Illumination Pipeline Overview

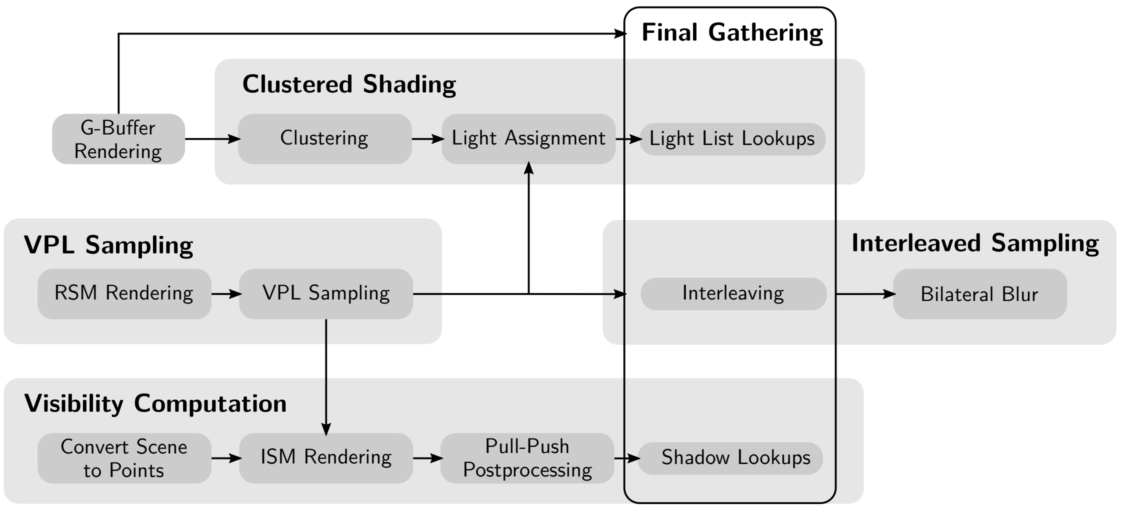

Figure 3.1: *The global illumination pipeline presented in this thesis. Light list lookups, interleaving and shadow lookups are some of the steps performed during final gathering.*

Figure 3.1 provides an overview of the global illumination pipeline presented in this thesis. Rendering the reflective shadow map and the subsequent VPL sampling (center row left in the diagram) will be covered in the following section. Section 3.3 describes the rendering process for imperfect shadow maps (lower row) in detail. The final gathering step uses data from the clustered shading technique (upper row, detailed in Section 3.5) and performs interleaved sampling (right part of middle row, detailed in Section 3.4).

During final gathering, clamping is used to remove singularities. Since this is a fairly trivial solution, it is not explained in further detail. In addition, the G-buffer rendering does not differ from common deferred rendering pipelines (Saito *et al.*, 1990) and is not examined in this thesis as well.

3.2 Virtual Point Light Sampling with Reflective Shadow Maps

Reflective shadow maps are often preferred as basis for VPL sampling techniques due to their simplicity and efficiency. Both of these advantages come from the fact that they are generated similarly to to conventional shadow maps, making them easy to implement and a perfect fit for GPUs. A downside is that they are well-suited only for computing the first light bounce.

Just like conventional shadow maps, RSMs render the scene from the viewpoint of a light. Besides the depth buffer used for shadow mapping, they render additional surface information, namely normal and color. As becomes apparent in Figure 3.2, this is similar to the additional G-buffers created by conventional deferred rendering pipelines. The result can be sampled to create VPLs with a certain position reconstructed from the depth buffer, and normal and color taken from the additional buffers.

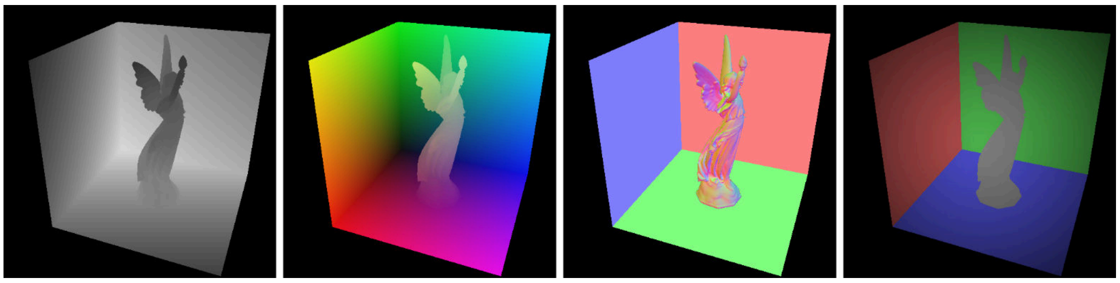

Figure 3.2: *A reflective shadow map as proposed by Dachsbacher et al. (2005). The buffers contain, from left to right, depth, world position, normal, and flux as a measure for light intensity. The flux buffer is used to account for different light intensities due to different solid angles of the pixels when using spot lights, and is not needed for directional lights. Figure reprinted from Dachsbacher et al. (2005).*

This thesis focuses on the later stages in the presented global illumination pipeline, therefore we use a rather basic approach for VPL sampling by creating an RSM and sampling it with a regular pattern. More advanced approaches are available (Section 2.5.1), some of which do not use RSMs (e. g., Hedman *et al.*, 2016)

3.3 Visibility Computation with Imperfect Shadow Maps

The original paper (Ritschel *et al.*, 2008) converts the scene geometry to a point set in a preprocessing step and uses the points to efficiently render hundreds of shadow maps in parallel. For performance reasons, they render only a small subset of all points per shadow map, relying on the large number of shadow maps, each using a different subset, to keep the overall error small (Figure 3.3). They use splatting to render the points into shadow maps and fill the resulting holes with a simplified version of the pull-push algorithm presented by Marroquim *et al.* (2007). The technique is also capable of multi-bounce indirect illumination.

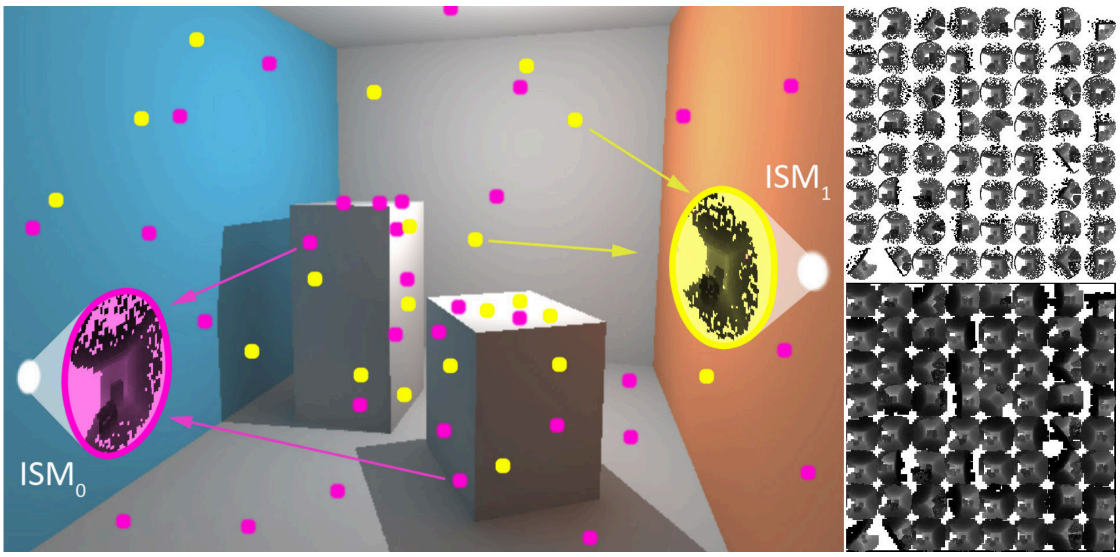

Figure 3.3: *Illustration of imperfect shadow maps. For each VPL, a different set of points, colored yellow and pink, representing the scene is chosen. These are rendered into one small shadow map per VPL (top right). A pull-push postprocessing algorithm fills holes in the shadow maps (bottom right). Figure reprinted from Ritschel et al. (2008).*

Ritschel *et al.* (2011) build on this approach by converting the scene into a triangle texture dynamically, and by sampling the points from that texture. Instead of computing a triangle texture, Barák *et al.* (2013) use the tessellation units of recent GPUs to dynamically convert triangles into points. We follow this approach since it integrates well into conventional rendering pipelines and by design supports dynamic geometry.

In contrast to Ritschel *et al.* (2008), we have found the pull-push postprocessing to be of little benefit when using regular point splatting. Instead, we demonstrate that it produces better results when applied to single-pixel "splats" as proposed in the original paper. Additionally, rendering points as single pixels opens interesting optimization opportunities.

The following subsections describe the regular point rendering process using splatting and subsequently detail the pull-push algorithm that is applied when rendering single-pixel points.

3.3.1 Point Rendering with Splatting

To convert the scene into points, all its triangles are first tessellated to meet a certain maximum size in order for the point conversion to not introduce extreme inaccuracies. Of the smaller tessellated triangles, the center and area are calculated and a point with matching center and area is created. While it might be more performant to create points just on the vertices of the triangles, at the same time it would be more inaccurate since this approach would enlarge the rendered area considerably over the triangle's extents.

Now a random VPL is chosen per point and backface culling is performed. Likewise, points that do not lie on the VPL's illuminated side are culled before splatting the point into the respective ISM. For performance reasons, the ISMs use a simple paraboloid projection in lieu of conventional cubemaps. Here, another advantage of using points comes into play: Compared to triangles, it is trivial to perform a paraboloid projection on them.

As mentioned above, we do not employ the pull-push postprocessing algorithm when rendering points with splatting. Instead, we simply enlarge all point splats by a constant factor, which results in similar quality but better performance.

3.3.2 Point Rendering with Pull-Push Postprocessing

As an alternative to point splat rendering, this subsection proposes a second approach that follows the algorithm of Marroquim *et al.* (2007) more closely with the intention of obtaining higher-quality shadow maps. More specifically, the points are rendered as a single pixel, albeit with additional attributes like size and normal. These are then used in a subsequent reconstruction pass to create an (ideally) hole-free shadow map (Figure 3.4). This approach also allows a point to be rendered into multiple shadow maps with a moderate performance impact, more on this issue follows in Section 4.4.3.

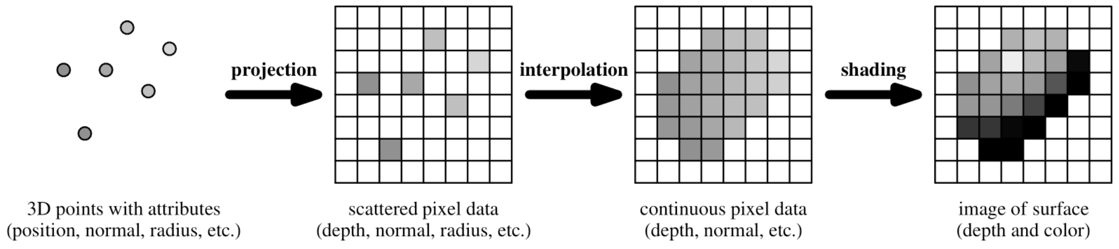

Figure 3.4: *Data flow of the pull-push surface reconstruction algorithm. Note that the shading step is skipped for rendering ISMs, as the depth information is sufficient. Figure reprinted from Marroquim et al. (2007).*

To provide a rough overview, the pull-push algorithm proposed by Marroquim *et al.* (2007) is a "pyramid method" (Strengert *et al.*, 2006) that uses the mipmap levels of a texture to first condense and then spread the data in the texture, in this case the sparse point set representing the scene. During the pull phase, it aggregates the information from four pixels of a finer level to one pixel of a coarser level. The subsequent push phase aggregates four pixels of a coarser level to one pixel of a finer level, but only if the pixel of the finer level contains no data or is determined to belong to an occluded surface. This way, closed surfaces are derived from single-pixel points.

Pull Phase

The pull phase reads four pixels of a finer level, computes a weighted average of them, and writes the result to one pixel of the next coarser level. However, it only considers pixels that pass the following tests: First, they need to contain a valid depth at all (i. e., a point must have been rendered into this pixel) and second, the point in this pixel must not be occluded. A point is considered to be occluded if it is outside the depth interval of the frontmost of the four pixels used for interpolation.

Figuratively speaking, the depth interval of a point denotes the range in which other points are considered to belong to the same surface and can be used for interpolation. For the rendered points, the depth interval is $[d; d + 2r]$ where d is the point's depth and r the point's radius. For interpolated points, the new depth interval spans the interpolated depth of the point to the highest depth value of the depth intervals of all "parent" points.

To calculate the attributes for a new point, the depth and radius of the parent points are interpolated in between with equal weights (Figure 3.5 (a) and (b)). Since the center of the aggregated point might not match the position of the pixel (this happens if less than all four pixels are used for interpolation), a displacement vector is also calculated and stored to accurately describe the point's location.

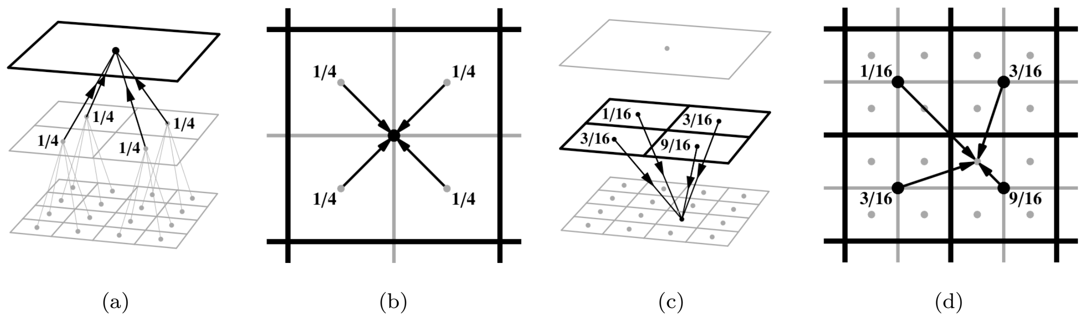

Figure 3.5: *Illustration of the weights used for interpolation during the pull phase ((a) and (b)) and push phase ((c) and (d)). Figures reprinted from Marroquim et al. (2007).*

Push Phase

Analogous to the pull phase, the push phase reads four pixels of the coarser level, rejects pixels that do not pass certain tests, and writes one output pixel into the finer level. First, just like in the pull phase, the input pixels must contain a valid depth and not be occluded to be considered. Additionally, a radius check is performed: If the pixel's location is outside the point's extents described by the displacement vector and its radius, then the point is not considered either. This is done to limit a point's influence to its actual size.

The target output location in the finer level might already contain data that was computed during the pull phase. This data is overwritten if its depth is behind the interpolated point's depth interval, i. e., if the original data is considered to be occluded. Otherwise, the original point is left untouched, as it is likely to be more accurate than the previously computed, interpolated point.

In contrast to the pull phase, the output pixel is not located in the center of the input pixels during the push phase. Therefore the weights used for interpolation are not uniform (Figure 3.5 (c) and (d)).

Marroquim *et al.* (2007) also use each point's normal to correctly limit the point's size, and Marroquim *et al.* (2008) propose Gaussian weights based on the pixel's distance to the point's location, instead of constant weights solely based on pixel locations. Both of these additions we have not implemented yet.

3.4 Interleaved Sampling

Interleaved sampling (Keller *et al.*, 2001) is often used to tremendously improve the performance of global illumination methods with a negligible quality impact. The general idea is as follows: Instead of calculating all samples per pixel, the samples are distributed over a block of pixels. Now, since each pixel processes only part of all samples, there is obviously information missing per pixel. Visually, this becomes apparent in the form of structured noise. To this end, a bilateral blur that preserves geometry edges similar to Laine *et al.* (2007) is used to spread the information and thereby even out the noise. The blur is often implemented in a separated fashion for performance reasons, although strictly speaking, bilateral filters are not separable.

Since this technique causes adjacent pixels to access disjoint sets of samples, naive implementations usually show bad cache coherence. As an improvement, Segovia *et al.* (2006b) proposed *non-interleaved shading of interleaved sample patterns*, or in short *de-interleaving*. They group those pixels that use the same sample set into smaller sub-buffers, coherently process each buffer, and then merge the sub-buffers to one large buffer again. See Figure 3.6 for the intermediate results of this algorithm.

Effectively, when applied to blocks of n^2 pixels, this technique cuts down the cost of sampling by $1n\ ^2$ at the expense of smoothing high-frequency changes in the samples over n pixels. Since global illumination is usually of low frequency, i. e., adjacent pixels usually receive a similar amount of indirect light, interleaved sampling does not significantly affect quality when applied to global illumination. However, this approach struggles when processing depth and normal discontinuities, where the bilateral blur has to ignore several pixels. Nevertheless, even in this case the quality degradation is hardly noticeable as soon as the global illumination results are blended over the rendered image (Section 5.5).

In most aspects we follow the standard approach for interleaved sampling. The only notable deviation is the implementation of de-interleaving that does not use any separate splitting and merging passes. For details, see Section 4.5.

3.5 Clustered and Tiled Deferred Shading

In real-time graphics applications, primarily video games, a technique called *clustered shading* (Olsson *et al.*, 2012), has seen increased usage. Its goal is to increase the efficiency of lighting a scene with multiple light sources. The effectiveness of clustered shading in the context of many-light global illumination has not been studied so far to our knowledge, and this thesis aims to contribute several data points to this end.

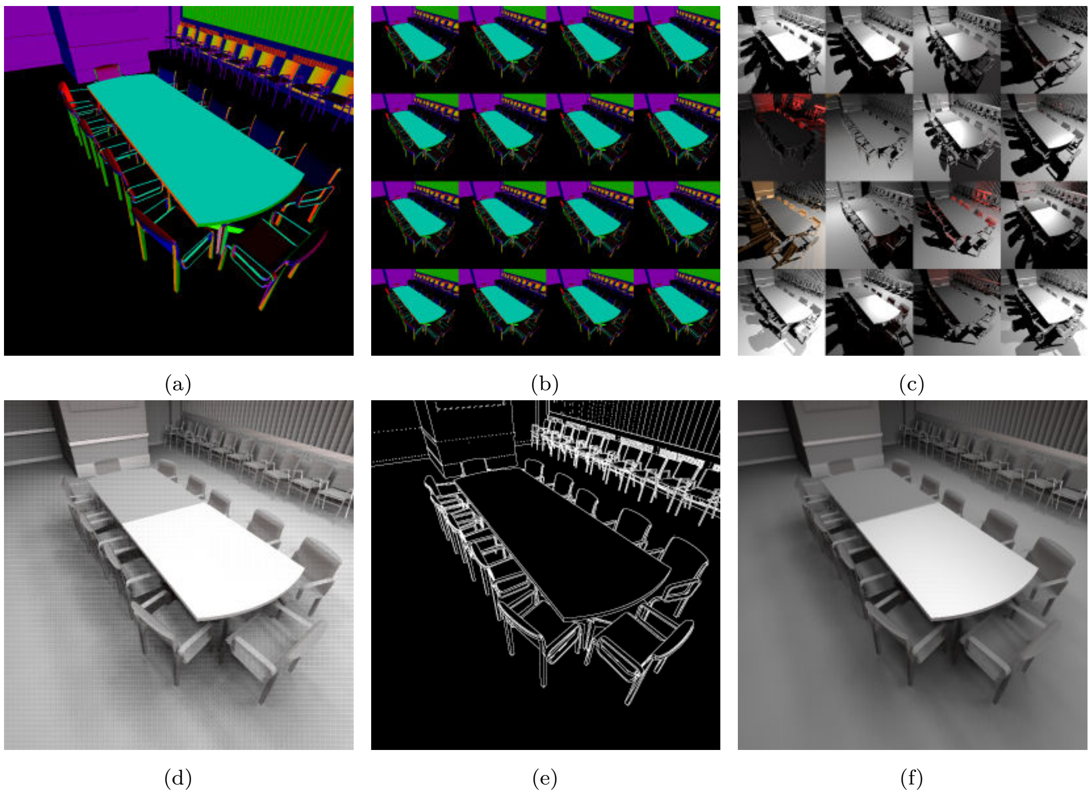

Figure 3.6: *Visualization of the de-interleaving process. (a) The original G-buffer. (b) The G-buffer is split into multiple smaller sub-buffers. (c) Each sub-buffer is processed using a different set of lights. (d) The sub-buffers are re-interleaved to one large buffer. (e) A depth discontinuity buffer is computed for the next step. This is usually omitted in modern implementations. (f) A bilateral blur is applied to mask the noise visible in (d). Figures reprinted from Segovia et al. (2006b).*

The clustered shading technique works as follows: First, the view frustum is divided into a fixed number of three-dimensional clusters. For each fragment, the respective cluster is determined with the purpose of ignoring all clusters with no fragments in them. Subsequently, each cluster is “assigned” a list of all lights that potentially illuminate fragments in that cluster, i. e., whose illuminated region is intersecting with a bounding volume of the cluster (Figure 3.7). All lights not intersecting a cluster are culled, i. e., not added to the cluster’s light list.

When shading a fragment, the algorithm first determines the fragment’s cluster again, and then iterates only over the lights in this cluster’s light list. This way, all the lights that have been discarded in the previous step do not need to be considered per fragment.

This technique yields the most gains when using lights with a limited radius. In that case, the culling is all the more effective since all lights that are farther away from a cluster than the light’s radius can be culled as well. However, many-light methods often

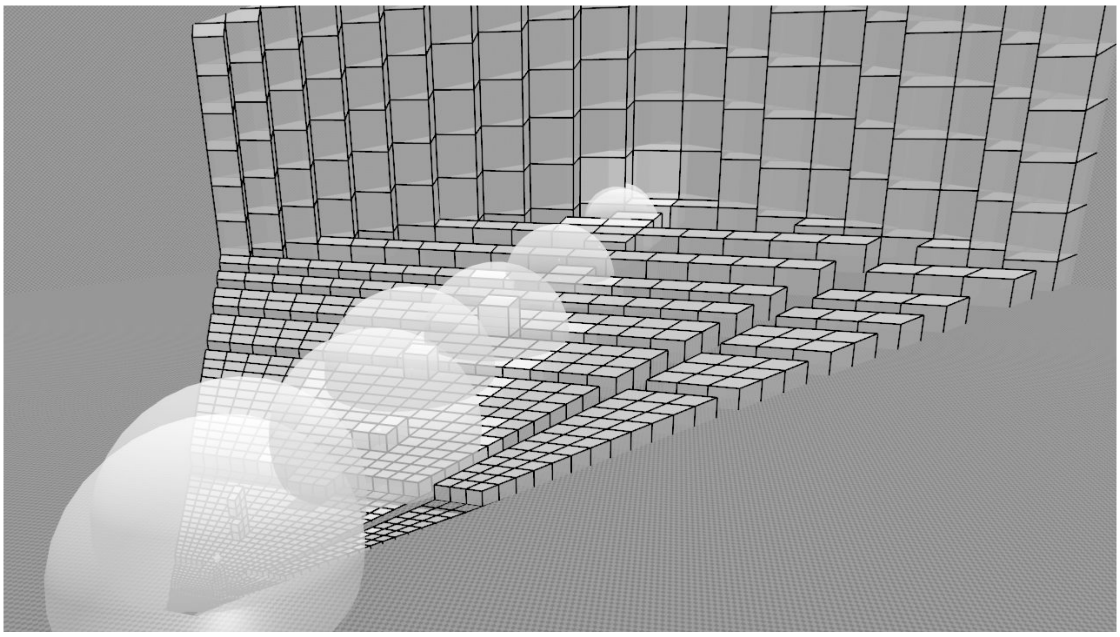

Figure 3.7: *Visualization of clusters (cubes) and light ranges (spheres) used for the light assignment step. Each cluster is assigned a list of all lights whose influence spheres are intersecting the cluster. Figure reprinted from Olsson et al. (2015).*

use infinite radii to enable light transport over long distances. Nonetheless, one can still expect to cull roughly half of all lights with this technique in a many-light context, since VPLs have a field of view of 180° instead of being completely spherical.

As an extension, Olsson *et al.* (2012) propose to calculate explicit bounds for the fragment's position in each cluster to enable more precise culling. We expected only moderate performance gains and have not implemented it. Another possible extension is to use the surface normal's direction as fourth dimension after the three spatial dimensions for clustering, allowing for a kind of backface culling for even greater efficiency.

Clustered shading is an extension and improvement of *tiled shading* (Olsson *et al.*, 2011), which uses screen-space tiles instead of three-dimensional view-space clusters. Tiled shading has previously been combined with many-light global illumination by Tokuyoshi *et al.* (2016). They use an order of magnitude more lights than we do, but stochastically limit them in range. As a result, the light culling is much more effective in their case. We implemented tiled shading as well and will compare it to clustered shading in Section 5.6.

Chapter 4

Implementation

This chapter will detail the implementation of the presented techniques. The next two sections will give an overview of the software as a whole and on the rendering pipeline, while the subsequent four sections describe the implementation of the RSM generation, ISM rendering, interleaved sampling, and clustered deferred shading.

The full source code is available in the project's repository[1] under the MIT license.

4.1 Utilized Frameworks and Libraries

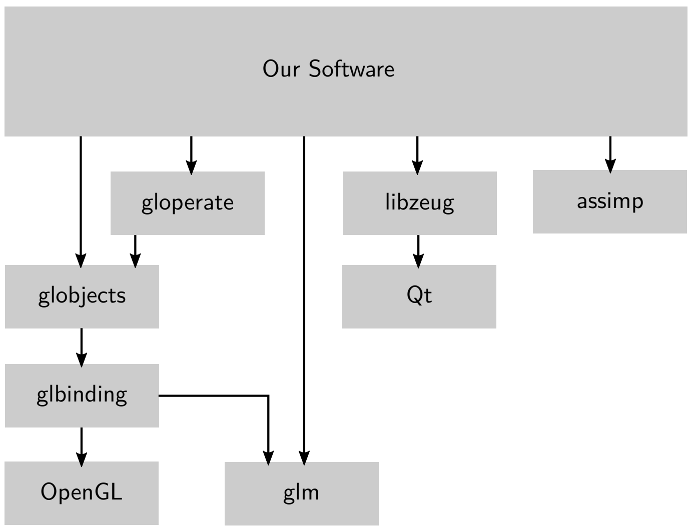

Figure 4.1: *An overview of the frameworks, libraries and APIs the presented software depends on.*

The implementation was built with C++ and OpenGL. It uses *gloperate*[2] as framework, which provides some basic functionality like navigation and resource handling. *gloperate* in turn is based on *Qt*[3], which is used for creating an OpenGL context and the UI. *libzeug*[4] is used to provide GUI widgets for quick manipulation of the rendering parameters.

[1]`https://github.com/karyon/many-lights-gi`
[2]`https://github.com/cginternals/gloperate`
[3]`https://www.qt.io/`
[4]`https://github.com/cginternals/libzeug`

globjects[5] provides a convenience abstraction layer around the OpenGL API, which is in some places directly accessed through *glbinding*[6]. For math functions and interaction with *globjects*, *glm*[7] is used. Additionally, *assimp*[8] is used to load the test scenes.

4.2 Rendering Pipeline Overview

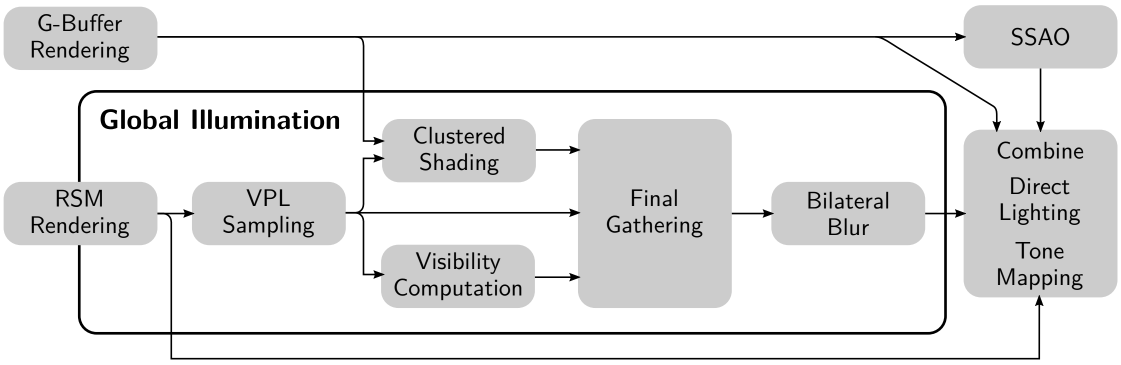

Figure 4.2: *Rendering pipeline overview. The global illumination section is a simplified version of Figure 3.1. The RSM rendering does not fully belong to the global illumination section, since it both provides a conventional shadow map for direct lighting, as well as additional information to form a RSM for VPL sampling.*

To provide some context, Figure 4.2 visualizes the complete rendering pipeline of the implemented application. It follows a standard deferred shading approach as it first renders geometry buffers with depth, normal and albedo information and uses these buffers to perform shading in a later stage. In the presented implementation, this stage is simply called "combine pass" and computes direct lighting, blends in the results of the SSAO postprocessing, and performs tone mapping that is necessary for HDR rendering. The result of the global illumination is a texture that contains indirect lighting, which is added to the direct lighting in the combine pass.

Most algorithms in this thesis are implemented using OpenGL compute shaders, while the G-Buffer, RSM, and the first stages of the ISM rendering use the regular fixed-function pipeline. The blur pass, SSAO and the combine pass utilize traditional screen-aligned triangles, since compute shaders would provide little benefit for their tasks. Due to the extensive use of compute shaders, OpenGL 4.3 is required to run the application.

[5]https://github.com/cginternals/globjects
[6]https://github.com/cginternals/glbinding
[7]http://glm.g-truc.net
[8]http://www.assimp.org/

4.3 RSM Generation and VPL Sampling

Reflective shadow maps (RSMs) provide a particularly efficient way to create VPLs, since they rely on regular rasterization. As such, the presented implementation of the RSM generation mostly uses the same code as the G-Buffer generation, barring a few modifications: First, normal maps are ignored and triangle normals are utilized instead, and instead of individual texture lookups during shading, the material's average color is used as diffuse color. Both these changes are made in order to avoid high-frequency normal and color changes in the RSM, which can only be captured correctly with a larger number of VPLs.

Additionally, the RSM is re-used as conventional shadow map. Usually, one might want to decouple RSM and conventional shadow map generation to use different resolutions or cascading schemes. It would also be possible to render the RSM with a less detailed version of the scene. Since the presented implementation uses no culling and LOD techniques, it was bottlenecked on geometry complexity. Rendering both RSM and conventional shadow map in one pass is the logical choice in this case. As a result, a variance buffer is rendered in addition to the regular G-buffers for use with the variance shadow mapping technique (Donnelly *et al.*, 2006).

Following the RSM generation, a compute shader samples the RSM in a regular pattern and writes the world-space position, normal and color of the sample points into a separate buffer. These information form the VPLs. Additionally, a second buffer is prepared that, instead of one three-component vector for position, normal, and color respectively, stores only one four-component vector in total, holding the VPL's position in the first three components and the normal packed into the last component (Cigolle *et al.* (2014)). This is done because the ISM rendering (Section 4.4) and light list calculation (Section 4.6) do not need the color information and especially the ISM rendering reads several VPLs per point, using a lot of bandwidth.

4.4 ISM Rendering

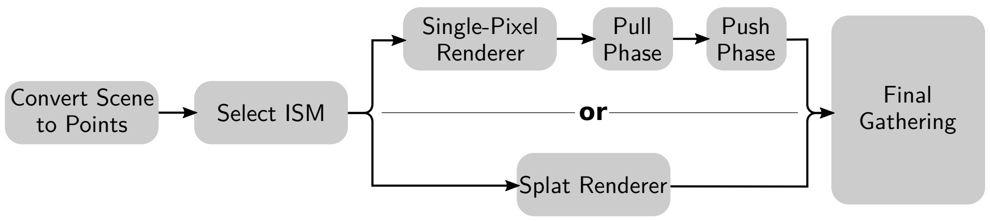

Figure 4.3: *Detailed pipeline of the ISM rendering process.*

Rendering imperfect shadow maps consists of several sub-stages; an overview is provided by Figure 4.3. First, a set of points is generated from the scene geometry. This process is detailed in the next subsection. Subsequently, for each imperfect shadow map a random subset of points is chosen and rendered using a paraboloid projection. To this end, two different approaches, the splat and single-pixel renderer, have been presented in the previous chapter. Their implementation is described in Sections 4.4.2 and 4.4.3. Subsequently, the pull-push postprocessing required by the single-pixel renderer is detailed.

4.4.1 Point Generation

We make use of OpenGL's tessellation shader to subdivide triangles to make them meet a certain maximum size. The tessellation control shader determines the tessellation levels according to the triangle's size. The tessellation evaluation shader does nothing more than correctly interpolating the vertex positions. The geometry shader, which now receives the smaller, tessellated triangles, converts each triangle to a point. The point's center equals the triangle's center, and the radius is chosen to make the areas of the point and triangle match. Note that no translation or projection has been applied yet, this is done in a later stage.

While a random subset of all points needs to be chosen for each ISM, special attention must be paid to ensure that each individual ISM uses the same subset of points across different frames. Otherwise, the ISMs will be temporally incoherent and cause flickering in the final rendering. To this end, each point's barycentric coordinate inside its triangle and the triangle's `gl_PrimitiveID`[9] are passed to a fast hashing algorithm. The result is used to determine an ISM ID (or, equivalently, a VPL ID) per point. How the ISM ID is used depends on whether point splatting or the single-pixel renderer is enabled.

4.4.2 Point Rendering with Splatting

In the case of point splatting, the geometry shader itself reads the VPL with the ID it determined, projects the point according to the VPL's data, performs culling, and discards the point if necessary, sets `gl_PointSize` to the correct size after projection, and emits a vertex. To prevent the GPU from assembling triangles from the geometry shader output and render point splats instead, the `output_primitive` must be set to `points` in the geometry shader's `layout`[10] definition. As explained previously, for point splatting all points are simply enlarged by a small constant factor in lieu of the pull-push postprocessing. Since only depth information are needed in this case, the fragment shader is empty.

4.4.3 Single-Pixel Point Rendering Using Compute Shaders

This alternative approach was inspired by the fact that the surface reconstruction algorithm proposed by Marroquim *et al.* (2007) works with single-pixel "splats". Since in this case the hardware rasterizer is not required to achieve good performance, this allowed for more flexibility and new optimization opportunities.

[9] https://www.opengl.org/sdk/docs/man4/html/gl_PrimitiveID.xhtml
[10] https://www.opengl.org/wiki/Geometry_Shader#Primitive_in.2Fout_specification

Approach

The general idea is to first let the geometry shader write the generated points into a buffer, which is then used by a separate compute shader pass to perform the actual rendering. However, in order to maintain temporal coherence, the points are not written into a single large buffer, as it would be hard to determine a location consistent across frames for each point in that buffer. Instead, the buffer is divided into one partition per VPL, and the points are written into the partition corresponding to the ID calculated before. An atomic counter that indicates the first free index is used per partition to sequentially fill these partitions without overwriting any data. The order of the points within each partition is undefined, but that does not change the output since they all are going to be rendered into the same ISM.

After the point buffer has been prepared, a compute shader is used to render the points. For each VPL, one work group reads through the respective partition of the point buffer and performs the transformation and culling against the VPL for each point. It then calls `imageAtomicMin`[11] to "render" the point with correct depth testing as a single pixel into the respective ISM. Since atomic operations operate on single channel integer textures only, it is not possible to use a second or third channel for additional attributes. Instead, one 32 bit channel is used, with 24 bits containing the depth, and 8 bits containing the radius. Since the depth is contained in the 24 more significant bits, the depth test still works correctly. Another version that used a 3D texture for additionally storing the normal was implemented as well, see the next subsection.

As an optimization, the compute shader renders each point into several ISMs. To this end, each work group reads the data of a fixed number of VPLs (e. g., 16) into shared memory. Then, while rendering the points, culling is performed per point on all 16 VPLs, and those VPLs where the point passes the culling test are collected into a local array. Thereafter, the point is transformed for each of the collected VPLs and rendered into the respective ISM. While this approach made the compute shader renderer more efficient, an attempt to implement the optimization for the splat renderer by emitting multiple vertices in the geometry shader resulted in a significant performance hit (Section 5.4.2).

Challenges when using a second render target

For a higher-quality postprocessing, the normals of the points are required. Since we did not use the normals in our final version, we were able to follow the simplified one-channel approach described above, but we had implemented a different version that also stored normals. To this end, the ISM texture was made three-dimensional, and the second index of the third dimension was used as additional "render target" to hold the radius and normal. The issues we faced during the development are described here.

In addition to roughly doubling the time needed to render the points due to the increased bandwidth, this approach introduces a second race condition besides the usual z-fighting that can occur. If two threads with differing depth values simultaneously

[11] https://www.opengl.org/sdk/docs/man/html/imageAtomicMin.xhtml

write into the depth buffer, using `imageAtomicMin` guarantees the correct value will end up in the buffer. However, if the higher value is written first and subsequently is overwritten by the second write, both writes have in fact succeeded. This results in both threads attempting to write into the second buffer, creating a race condition since the execution order of the writes is undefined. This problem can be alleviated by adding a synchronization point after the atomic write and by then reading the memory location that has just been written to. If it is equal to the written value, one can assume the write actually succeeded and was not overwritten by a second, simultaneous write. Only then the write to the second buffer is performed.

This solution had little cost (12 % of the point rendering or 3 % of the total ISM rendering time), but it fixes the race condition only inside one work group and not between work groups, for which there are no efficient synchronization primitives on the GPU. On a NVIDIA GTX 750 Ti, the race condition was still observable but acceptable. Interestingly, on a GTX 980, the race condition seemed not to occur at all with the described workaround.

Günther *et al.* (2013) propose to double the texture size of the depth buffer and use each second texel as lock with the help of OpenCL's `atomic_cmpxchg`[12] function (`atomicCompSwap`[13] in GLSL) to write both depth and additional attributes without race conditions. A more elaborate solution that might open even more optimization opportunities would involve a more advanced software renderer, e. g., following the approaches of Laine *et al.* (2011).

4.4.4 Pull-Push Postprocessing

Our implementation of the pull-push algorithm to reconstruct surfaces from single-pixel points is relatively straightforward. For each miplevel to be calculated, a compute shader is launched via `dispatchCompute`[14]. Since the first step of the pull phase has different inputs than the subsequent steps (its only inputs are depth and size, not the displacement vector and depth interval), a slightly different shader is used there. Analogously, the last step of the push phase outputs only depth, therefore a shader variant is created for that step as well. For the same reason, the initial input texture (the "render target" used by the single-pixel renderer) and final output texture (the finished ISM used later for shadowing) have different formats than the mipmap levels used by the pull-push algorithm.

In the pull phase, one compute shader invocation calculates one output pixel. It reads its four input pixels, determines which of them are valid, and uses the valid ones with equal weights for interpolating.

During the push phase, groups of four output pixels share the same input pixels. This can be exploited by letting each shader invocation compute four output pixels while still reading only four input pixels. Naturally, the weights applied to the input pixels

[12] https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/atomic_cmpxchg.html
[13] https://www.opengl.org/sdk/docs/man/html/atomicCompSwap.xhtml
[14] https://www.opengl.org/sdk/docs/man4/html/glDispatchCompute.xhtml

need to be adjusted per output pixel. For invalidating input pixels, their weights can simply be set to zero instead of separately marking them as invalid.

Note that there is still lots of room for optimization. Especially the pull phase is basically a parallel reduction which is well-researched and for which optimized algorithms on the GPU are available (Harris, 2007). Several optimizations from that field might be applicable here.

4.5 Interleaved Sampling Using Compute Shaders

The presented implementation performs interleaved sampling, that is, it processes only a fraction of all VPLs per pixel and uses a blur pass to mask the resulting noise. Since adjacent pixels now process different sets of VPLs, this approach results in low cache coherence. To this end, de-interleaving is employed (Segovia *et al.*, 2006b).

Often, de-interleaving is implemented by splitting the G-buffer into several smaller buffers, each containing all pixels with the same sample set. Each buffer is processed with its respective sample set, and then the buffers are re-interleaved into a large G-Buffer.

We propose to perform de-interleaving, sample processing, and re-interleaving in a single pass using compute shaders. Just like the original method, this approach allows for cache-efficient processing since each work group processes a set of pixels that have the same set of VPLs assigned. However, instead of splitting the G-Buffer and then re-interleaving it to perform coherent reads and writes, we propose to let the compute shader read and write directly to the respective pixels in the G-Buffer.

There are several reasons for this approach: First, implementation complexity is reduced, since the buffer splitting and re-interleaving phases are not needed as separate shader passes anymore. Second, as another consequence of eliminating these phases, the total amount of memory read and written is greatly reduced, potentially saving bandwidth. And third, no additional storage is needed.

A potential downside is that this technique performs one scattered read of the G-Buffers, compared to several coherent reads performed by the buffer splitting method. However, the bottleneck of the final gathering phase lies within the loop over the VPLs, therefore an initial one-time scattered read is likely to not have any impact but to be swallowed by latency-covering techniques of the GPU. Theoretically, the work groups could be distributed over the GPU's processor clusters (Streaming Multiprocessors on NVIDIA, Compute Units on AMD) in a way that adjacent pixels are still processed on the same cluster, but unfortunately, application developers have no control over this.

Since only a subset of all samples is processed per pixel and this subset repeats every few pixels, interleaved sampling results in structured noise. To this end, a bilateral blur similar to Laine *et al.* (2007) is often used. The blur filter is bilateral, i. e., it takes depth and normal discontinuities into account, to prevent light from bleeding over geometry edges. Although bilateral filters are in fact not separable, they are commonly implemented in a separated fashion. This trades a negligible quality impact for large performance benefits, and the presented implementation follows this approach.

While the bilateral blur achieves close to perfect results on even surfaces, the structured noise is harder to mask along geometry edges due to missing information. This problem becomes even more apparent when using regular VPL sampling, which makes each pixel process VPLs that are near each other. For example, this might result in the first pixel inside a 4x4 block processing VPLs that are near the pixel's location, while the second pixel processes VPLs that are all farther away, causing the differences between the pixels to be amplified. To this end, the order of the VPLs in the VPL buffer created during the VPL sampling phase is simply shuffled, using a fixed permutation precomputed on the CPU. See Section 5.5 for the effects of this small optimization.

Instead of shuffling, an interesting and more elaborate solution to make the noise easier to mask are Metalights (Faure *et al.*, 2010). They sort the VPLs after their contribution to the image and pick a representative subset per pixel. While the blur is still necessary, this might provide better results for problem cases such as thin structures.

4.6 Clustered and Tiled Deferred Shading

Since both tiled and clustered deferred shading discard lights only if they do not contribute any light energy to a cluster of pixels, they never alter the rendered output when implemented correctly and purely affect performance. Nevertheless, there are several implementation choices to be made.

4.6.1 Clustered Deferred Shading

The presented implementation of clustered deferred shading uses 128 pixels as screen-space tile size and 16 depth slices. It also draws on an optimization proposed by Persson *et al.* (2013) by using a larger near cluster for better depth slice utilization.

In some situations it is advantageous to perform the light assignment step on the CPU as done by Persson *et al.* (2013). In their context of computer games, they have mostly small light radii, making it possible to quickly determine the clusters that are reached by a certain light. For global illumination, however, infinite light radii are preferred, resulting in a higher number of clusters that are reached by each light source. Therefore, starting with all clusters that contain fragments and iterating over all lights for each cluster becomes viable, as opposed to determining the set of reached clusters per light. The more uniform control flow of this approach makes it a reasonable choice to use GPUs for the light assignment step. Additionally, since the lights are already generated on the GPU during the RSM sampling phase, CPU-side culling would require copying light data to the CPU and culling results back; performing culling on the GPU eliminates these data transfers.

The clustered shading technique is divided into three phases: clustering, light assignment and shading. Each of these corresponds to one compute shader dispatch call.

Clustering. In this step, each work group processes one 128^2 px screen-space tile. Since the number of threads in a work group is limited, each thread in the work group iterates over several fragments. For each fragment, the corresponding depth slice is determined and marked as used in a shared array of 16 booleans. Since no thread writes False or reads from the shared array, no data races can occur and no synchronization is needed in this step.

Now the work group counts the number of used depth slices. This can be done with 16 threads, utilizing atomic increment operations on a shared variable, or one thread that iterates over the sixteen booleans. Since both ways take a negligible amount of time, the choice does not matter in practice.

The number of used depth slices is then added to one global atomic counter. The corresponding GLSL function (`atomicAdd`[15]) returns the value of the counter before the addition. This way, each work group "allocates" some space in a global list of used clusters. It then writes an ID identifying each cluster into the allocated section in that list. To be able to later determine a cluster's position in the global list, a 3D texture with one texel per cluster is filled with each cluster's location.

Instead of a compact list of used clusters and a compact texture of light lists, one could omit the list of used clusters and employ a sparse texture of light lists where the light list of each cluster has a fixed position. However, this would obviously require more memory, even if only a few megabytes, since most parts of the light list texture would be unused. It would also make the light assignment step more inefficient, since it would become necessary to test for each cluster whether it is used, instead of simply processing a compact list of used clusters.

Light assignment. With a new dispatch call, each thread now performs culling for one light and one cluster. Given the cluster coordinates, it reconstructs the cluster's bounds and checks on which side of the plane described by the VPL's position and normal the cluster is located. More specifically, if any of the cluster's eight corners is on the illuminated side of the VPL, the light cannot be culled and is added to the light list of the cluster.

The light lists are stored in a 2D texture with a height of the maximum amount of clusters and a width of the number of VPLs. The nth row in that texture contains the light list of the cluster at index n in the global list of used clusters created in the previous step. Additionally, each light list is divided into 16 partitions, one partition for each pixel in an interleaving block. See Figure 4.4 for a visualization of this texture. The partitioning not only simplifies the light list lookup during shading, it also guarantees that each light is processed exactly once in an interleaving block.

As a simplified example, suppose that only four lights are used and the interleaving block size is 2x1, i. e., that it contains only two pixels. As described in the interleaved sampling algorithm, in this case the first pixel will process the first half of its light list, while the second pixel will process the second half of its respective light list. If the first pixel's light list contains lights 3 and 4, while the other pixel is in a different cluster whose light list contains lights 2 and 3, then for both pixels only light 3 will be processed. As a result, one light will have been accounted for twice, while two other lights have not been considered at all. This was visible as darkened or brightened lines along the screen-space

[15] https://www.opengl.org/sdk/docs/man/html/atomicAdd.xhtml

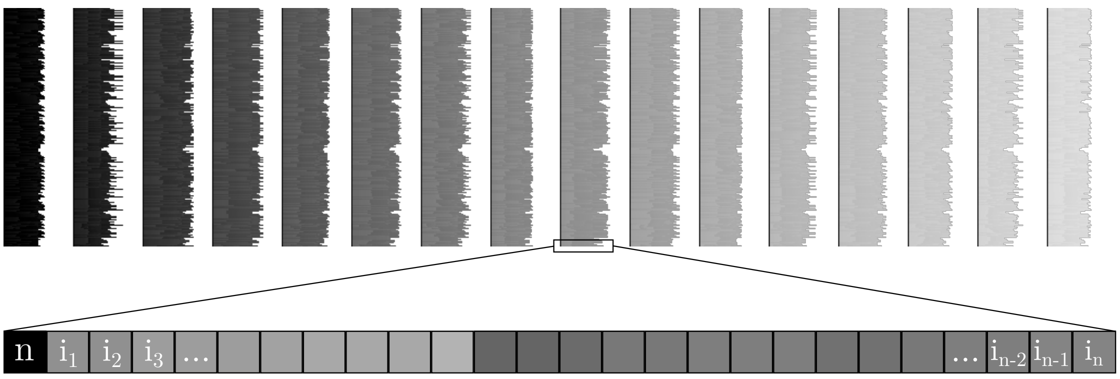

Figure 4.4: *Visualization of the light lists texture. Each pixel row contains the light list of one cluster. Each gray pixel represents an index to an VPL, a lighter gray is a higher index. White pixels contain no data. The lists are divided into 16 partitions, one for each pixel in an interleaving block. One such partition is visualized with more contrast at the bottom. The first value in each partition is the number of lights in that partition, visible as dark line in front of the partitions. Note that the indices in a partition are not necessarily continuous, as they are written by different shader invocations, whose order of execution is undefined.*

tiles used by the clustered shading algorithm. The partitioning scheme completely fixes this by guaranteeing that the first pixel always processes lights 1 and 2, while the second pixel always processes lights 3 and 4, minus culling results respectively.

While the partitions cannot reasonably be shrinked much, one could estimate the average of used depth slices per tile to reduce memory consumption. Since the list of used clusters is stored in a compacted list as discussed previously, individual tiles can still use more depth slices than estimated and not break the algorithm. An average of 4 out of 16 depth slices is likely enough for most scenes and would cut down the memory consumption of the light list texture by 75 %. To provide a reference: Since each partition contains only 64 lights, 8-bit indices are sufficient. The light list of each cluster thus needs 1 KB. For a full HD resolution and screen-space tiles of 128 px, the maximum number of clusters is 2160, thus a little over 2 MB of memory is needed. With the discussed estimate of 4 used clusters per tile, this can be cut down to 540 KB.

Shading. The final gathering shader processes one pixel per invocation. Without clustered shading, each invocation simply processes the subset of VPL's that should be used as determined by the interleaved sampling algorithm. With clustered shading, it will instead first determine the pixel's cluster and subsequently process the lights in this cluster's light list and in the partition corresponding to the pixel's position in its interleaving block.

4.6.2 Tiled Deferred Shading

In addition to clustered deferred shading, we also implemented its predecessor tiled shading. Since tiled shading does not perform clustering in the z-dimension (or in other words, it uses only one depth slice per tile), it allows for a simpler implementation.

We integrated tiled shading directly into the final gathering shader, in an approach very similar to Andersson (2011). As a reminder, during final gathering each work group processes those pixels from several interleaving blocks that process the same set of VPLs, and each invocation processes one pixel. With tiled shading, each pixel's z-value is used to build a minimum and maximum z-value for this work group, utilizing atomic operations. In a second step, each invocation processes one VPL and culls it against the bounding box of the pixels, which is defined by the pixel's minimum and maximum fragment coordinates and z-values determined in the previous step. All lights that pass the culling test are added to a light list in a shared array, again utilizing atomic counters to ensure no gaps are produced and no light is overwritten. Subsequently, final gathering is performed as usual with the difference of reading from the light list in the shared array instead of a light list computed in a previous shader pass.

Similar to the integrated interleaved sampling algorithm, this implementation has several benefits. Again the implementation complexity is reduced, since no additional shader passes and no additional buffers with corresponding data layout considerations are needed. Second, no additional storage is needed, and third, this implementation, although simpler, turned out to be faster than clustered deferred shading. See Section 5.6 for details.

Chapter 5

Results and Discussion

This chapter details the performance and rendering quality characteristics of the implemented techniques and examines their memory usage. For each technique, it will subsequently discuss the up- and downsides.

5.1 Scenes, Settings and Testing System

The implementation is tested in the Crytek Sponza and San Miguel scene, with 260k and 7.9M triangles respectively, both provided by McGuire (2011). Unless otherwise noted, the parameters used for the measurements and screenshots are:

- 1920x1080 px output resolution
- 1024 VPLs
- 2048^2 px ISM texture, i. e., 64^2 px per ISM
- Single-pixel point renderer enabled, 16 VPLs considered per point, up to 4 collected
- 4x4 interleaving pattern
- Tiled shading enabled

The hardware specification of the testing system is as follows:

- Intel Xeon W3530 with four cores at 2.8 GHz
- NVIDIA GeForce GTX 980 factory-overclocked by ca. 4%, driver version 375.57

The GTX 980 has been locked to its base clock rate of 1164 MHz using SetStablePowerState.exe[1], preventing its GPU Boost feature[2] from altering the results depending on the GPUs temperature and power draw. This comes at the cost of preventing the GPU from using its maximum potential, but combined with the factory-overclocking, the results should be close to the real-world performance.

[1]https://developer.nvidia.com/setstablepowerstateexe-%20disabling%20-gpu-boost-windows-10-getting-more-deterministic-timestamp-queries

[2]http://www.geforce.com/hardware/technology/gpu-boost-2

5.2 Rendering Time Breakdown

Table 5.1: *Timing breakdown of an entire frame rendered by the presented software. Timings are in milliseconds.*

G-Buffer	RSM	ISM	GI	SSAO	Combine	Blit	Total
0.6 ms	0.2 ms	2.7 ms	3.2 ms	1.4 ms	0.3 ms	0.3 ms	8.70 ms

Table 5.1 gives an overview of the time needed for rendering a single frame. The Crytek Sponza scene and default settings have been used. All timings measure GPU time only; there are no noteworthy calculations done on the CPU. With a total of 8.7 ms, the application achieves real-time frame rates while leaving room for more expensive calculations, higher output resolutions, or more detailed scenes.

The GI, SSAO, and combine pass operate in screen space and are mostly dependent on the screen size (and, of course, on light and sample count in the case of GI and SSAO, respectively). The G-Buffer and RSM generation follow the usual performance characteristics and are mainly dependent on output resolution and geometric complexity. The performance behavior of the ISM rendering heavily depends on the settings and is further detailed below. The blit phase copies a texture to the back buffer and serves debugging purposes only.

5.3 RSM Generation and VPL Sampling

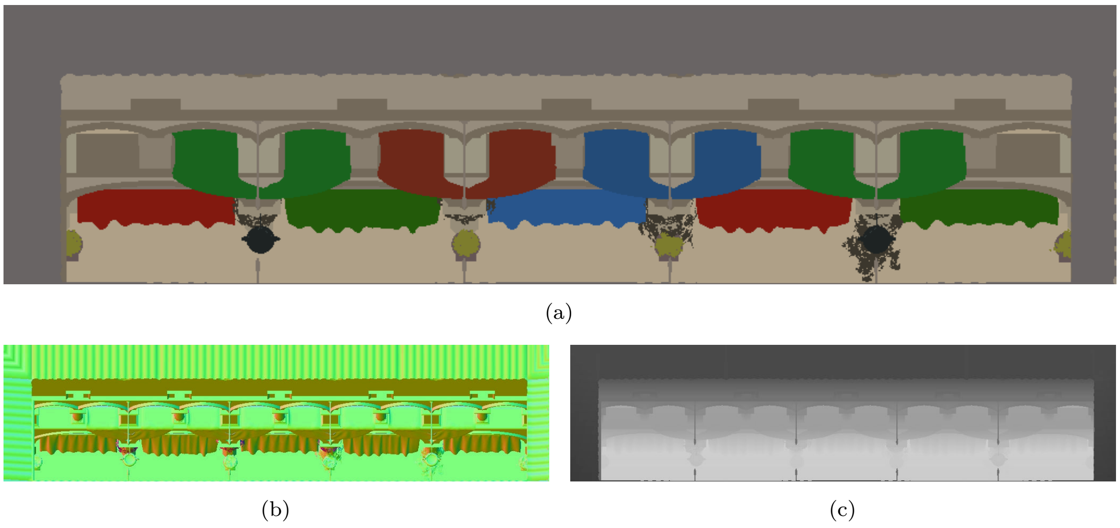
(a)
(b)
(c)

Figure 5.1: *A reflective shadow map, with its (a) diffuse, (b) normal, and (c) depth buffer.*

As explained before, the RSM generation re-uses most of the code of the G-Buffer generation; see Figure 5.1 for an impression on the rendered result. Accordingly, the performance characteristics are similar to the regular geometry pass of deferred renderers.

Since the implemented VPL sampling distributes the samples over the entire shadow map, we simply limited the VPLs' locations to the relevant area by tuning the light's extents. As a result, an aspect ratio chosen specifically for each test scene is used. Since our implementation re-uses the RSM as shadow map for direct lighting, we chose a resolution of 512^2 px (or, in the case of the Sponza scene, 1024x256 px), which is more than required by the naive VPL sampling. The RSM is rendered in 0.17 ms for the Sponza scene.

The VPL sampling itself takes only several microseconds and is negligible. Bear in mind that, in order to achieve high quality levels, a more elaborate sampling algorithm needs to be implemented, which can in fact take most of the available time. For instance, the implemented sampling does not take the relevance of the sampled VPLs to the current frame into account. This causes the entire system to be rather inefficient because it chooses lights that might contribute little or nothing to the rendered output (Figure 5.2). An additional downside of the naive sampling is the poor temporal stability when the scene light moves. This is due to each VPL staying at the exact same position in the light's viewport; by consequence their positions strictly follow the light's movement, jumping over depth discontinuities along the way.

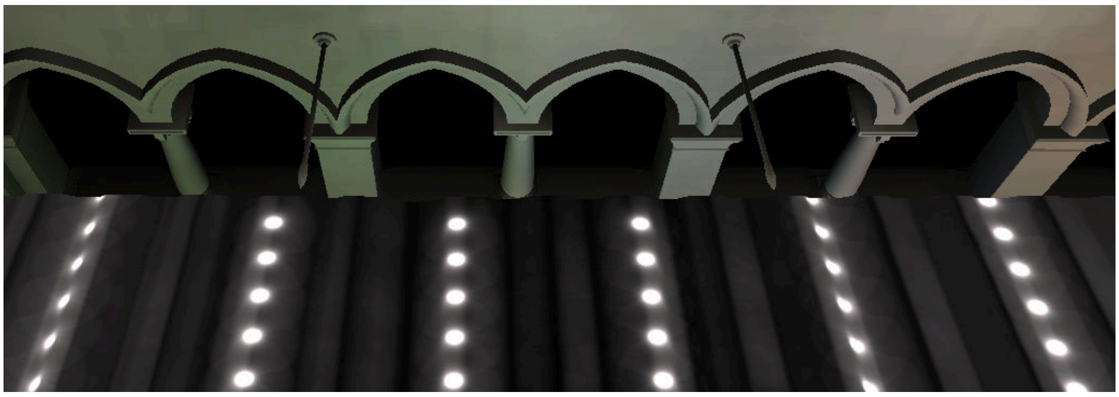

Figure 5.2: *A visualization of inefficiently placed VPLs. Each of the bright spots is a VPL. Here, they are all placed on the roof pointing upwards. Since there is no further geometry above them that could be illuminated, they do not contribute to the final image.*

5.4 ISM Rendering

This section presents the results for visibility testing using imperfect shadow maps. The next three subsections will show the outcome in terms of quality, performance, and memory usage. The technique does not handle high geometric complexity of the test scenes well; this is detailed in the fourth subsection. Afterwards, the two different ISM renderers are compared and the suitability of ISMs for global illumination in general is discussed.

5.4.1 Quality

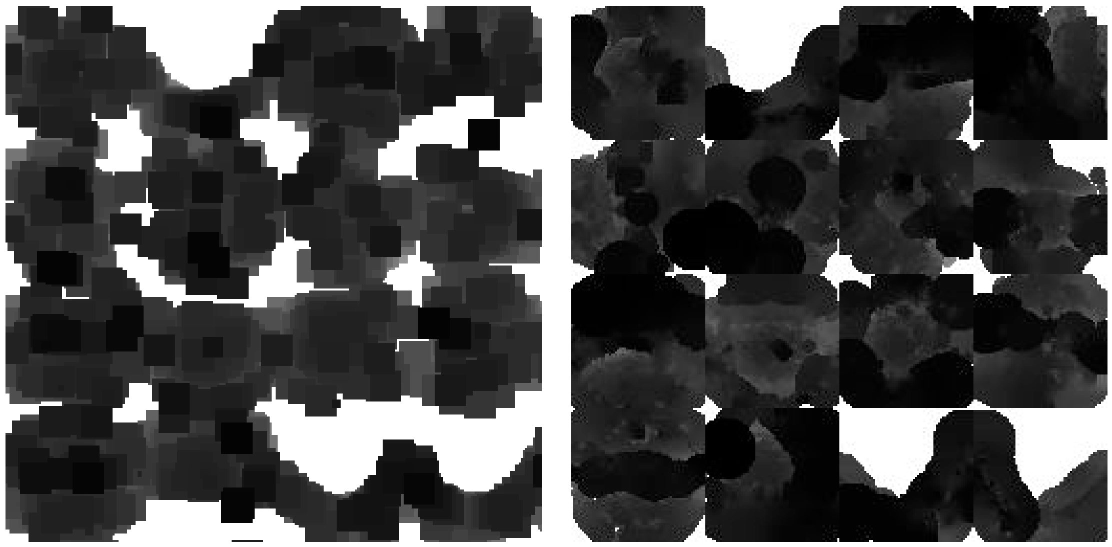

Figure 5.3: *ISMs rendered using the splat renderer (left) and single-pixel renderer (right) with default settings. The single-pixel renderer performs interpolation between points, renders more points, and does not let points bleed into neighboring ISMs, but takes more time.*

Figure 5.3 shows a few ISMs rendered with the splat and single-pixel renderer. The imperfections do show, and not only through the low resolutions. The distortion of the surface silhouettes by the splat renderer or postprocessing contribute their part, making it hard to identify which part of the Crytek Sponza is rendered. However, keep in mind that the imperfections, while being noticeable in each individual ISM, are expected to partly cancel each other out and result in an acceptable average error in the final rendering.

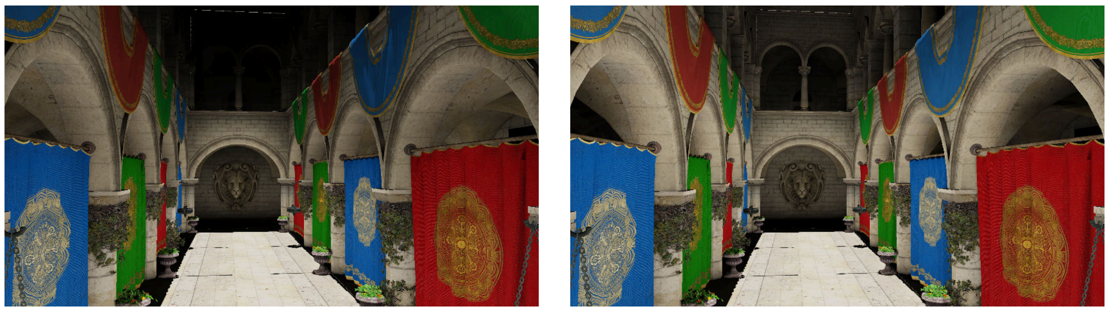

Figure 5.4: *Darkening caused by the splat renderer (left) compared to the single-pixel renderer (right). Note that there is no skylight rendered, which leads to an unnaturally dark upper part in the image even for the single-pixel renderer.*

Comparing the screenshots in Figure 5.4, it becomes apparent that the splat renderer causes visible darkening in the upper part of the image. The reason is probably that the point splats are always oriented towards the camera and do not take the point's normal into account when rendering, amplifying the usual aliasing artifacts of common shadow maps. As a result, any point size larger than one pixel causes surfaces that are not directly facing the camera to appear nearer than they actually are when doing shadow lookups in ISMs. A larger shadow bias could compensate for that but would introduce heavier light leaking. Another possibility is to use the normal to calculate a point's depth per fragment at a potentially high performance cost.

As the single-pixel renderer does not use splats, this problem is largely mitigated. In a certain way it performs the per-fragment depth calculation implicitly during interpolation in the postprocessing phase.

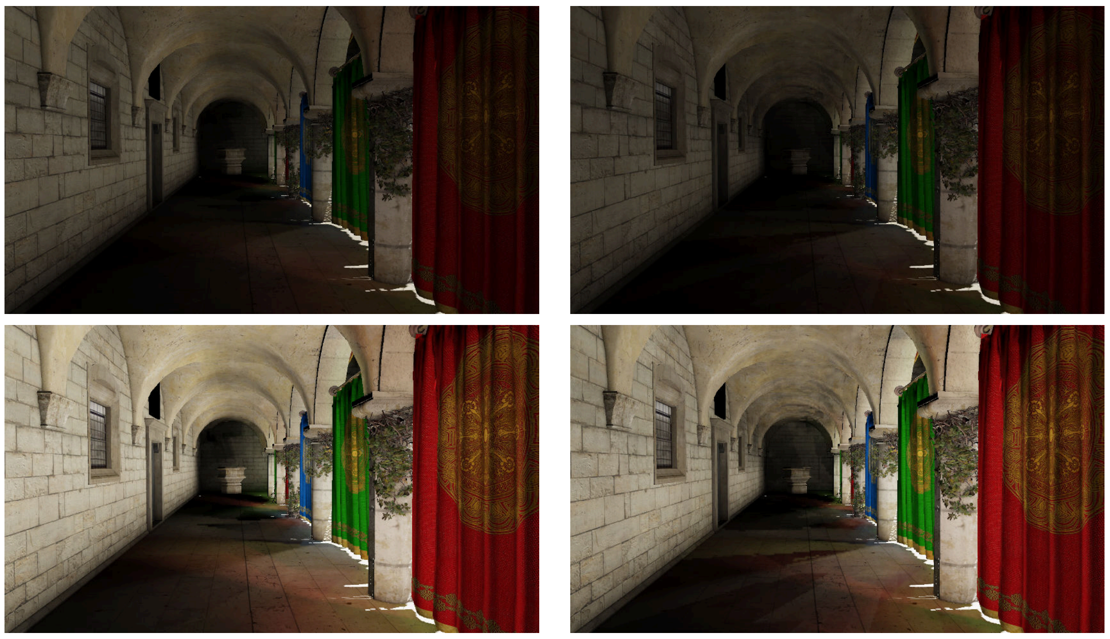
Figure 5.5: *Light leaks caused by imperfect shadow maps rendered with the splat renderer (left) and single-pixel renderer (right). VPLs from beneath the curtains leak light onto the wall and ceiling, while VPLs on the pillars and curtains light the floor. The renderings in the bottom row were rendered with a higher exposure for illustration.*

Figure 5.5 shows a case which the ISM technique has difficulties with. The image should be mostly dark or at least uniformly lit through the small gap below the curtains, instead the wall and ceiling appear too bright and the floor displays several artifacts. The root cause is that the VPLs are placed right behind the curtains, because of which the occluding geometry, i. e., the curtain, is very near to the light source. Due to the randomness involved when selecting point sets for rendering ISMs, it often happens that the points near to the light source are rendered into other ISMs, leaving a large hole behind. The single-pixel renderer fares a bit better than the splat renderer since it uses more points, but does not provide a satisfactory result either.

Our implementation hides some of the inaccuracies of ISMs by using a relatively large shadow bias. The resulting light leaks are shown in Figure 5.6. While the splat

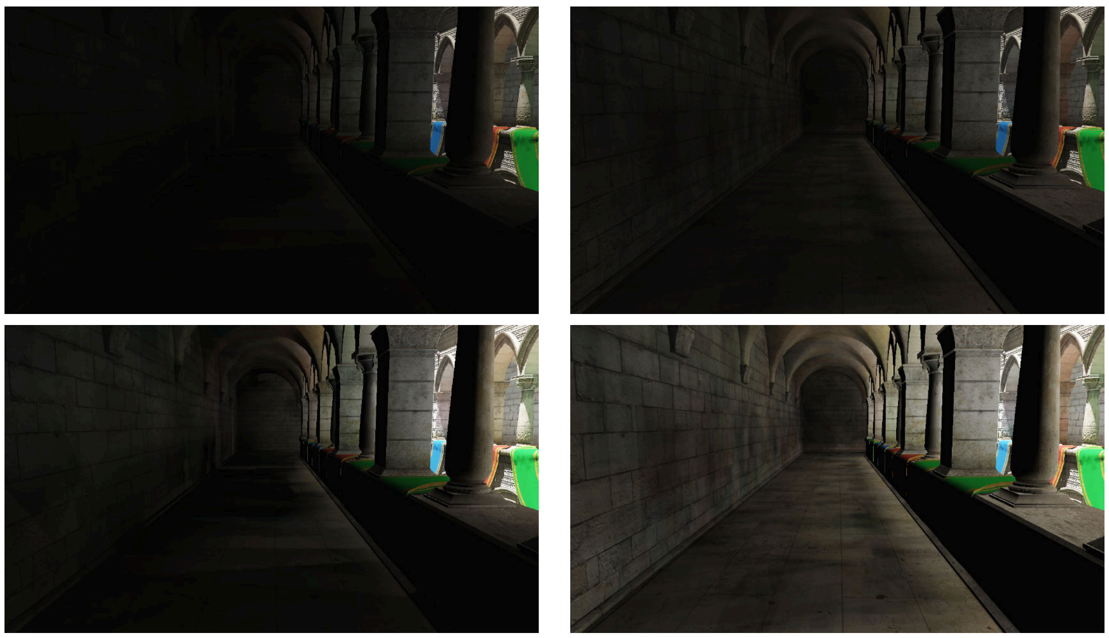

Figure 5.6: *Light leaks caused by using a relatively large shadow bias with the purpose to hide artifacts of the ISMs. The screenshots to the left use the splat renderer, the right ones the single-pixel renderer. The bottom row was rendered with a higher exposure.*

renderer fares better here, it does so mainly by darkening the whole image since it uses rather large points. The single-pixel renderer on the other hand correctly lets light illuminate the left wall through the columns, but shows a rather noisy result.

5.4.2 Performance

Table 5.2: *Timings of the ISM renderers with different settings.*

Splat Default Settings	Single-Pixel Default Settings	Splat with Single-Pixel Settings	Single-Pixel with Splat Settings
1.08 ms	2.74 ms	10.14 ms	2.31 ms

Table 5.2 shows the time needed for rendering the ISMs with both renderers and different settings. Note that the comparison between the two renderers with default settings is not a fair one, since the single-pixel renderer renders more points and by default clamps each point's extents to its ISM. If the splat renderer is changed to behave similarly, it takes 0.3 additional milliseconds for clamping, and 8.66 additional milliseconds for using the same technique to collect VPLs that the single-pixel renderer uses.

There are several reasons for this heavy slowdown: First, this is implemented by emitting multiple vertices in the geometry shader, which is known to have poor performance on current GPUs. Second, the additional fillrate and overdraw became an issue when rendering too many points, a bottleneck that is unlikely to occur when using the single-pixel renderer. And third, while the single-pixel renderer uses shared memory to load the 16 VPLs once, the geometry shader of the splat renderer has no access to shared memory and has to load all 16 VPLs per invocation, creating high register pressure.

Conversely, if the single-pixel renderer does not perform clamping, the push phase needs 0.07 ms less (0.85 ms to 0.78 ms), and when it considers only one VPL per point, the point renderer takes 0.41 ms less (0.65 ms to 0.24 ms).

Since the presented implementation uses no adaptivity, these numbers are independent from the viewport. They are however slightly affected by VPL placement, since it influences culling. In a second scenario, where the sunlight shines directly from above, all VPLs are placed on the floor facing up. As a result, much fewer points are culled during ISM rendering, resulting in slightly higher timings. Both renderers need about 0.1 ms more in this case.

Detailed Performance Measurements for the Single-Pixel Point Renderer

Table 5.3: *Timing breakdown of the single-pixel point renderer.*

Point Collection	Point Rendering	Pull Phase	Push Phase	Total
0.83 ms	0.65 ms	0.39 ms	0.85 ms	2.72 ms

Table 5.4: *Timing breakdown of the pull (PL) and push (PS) phase. The numbers of the individual steps indicate to which mipmap level they write, which is why the pull phase starts at 1 and the push phase has descending numbers. All timings are in milliseconds.*

PL 1	PL 2	PL 3	PL > 3	PS > 2	PS 2	PS 1	PS 0	PL Total	PS Total	Total
0.16	0.16	0.04	0.03	0.06	0.08	0.30	0.41	0.39	0.85	1.24

Table 5.3 gives more detailed performance measurements of the single-pixel renderer, while Table 5.4 further breaks down the individual steps of the pull and push phase. Note how the timings scale roughly as expected (PL 3 and PS 2 are taking roughly four times longer than PL 2 and PS 1, respectively), except for the first and last step. This is due to the reduced input data size (or output size, respectively), as explained in Section 4.4.4. Also note how the push phase does take roughly the time of the pull phase if it were not for the last phase, where it writes to the full 2048^2 px of miplevel zero, whereas the largest miplevel written to by the pull phase is level one with 1024^2 px.

5.4.3 Memory Usage

The point splat renderer requires no more memory than the ISM texture itself requires, which is 8 MB for a 2048^2 px 16-bit depth buffer. The single-pixel renderer requires additional memory. First, it uses a buffer for storing the points. This is implemented as four-channel 32-bit float texture, with the first three channels containing the position, and the last channel containing radius (8 bit) and normal (24 bit, see Cigolle *et al.*, 2014). With a maximum point count of 2048 per ISM (keep in mind they are rendered into multiple ISMs later), this buffer uses 32 MB. The additional textures created are the render target of the single-pixel renderer (single-channel 2048x2048 px, 32-bit integer, 16 MB), the mipmap levels used by the pull-push algorithm (four channels, 32-bit, approx. 22 MB) and the final ISM (same format as used by the splat renderer, 8MB).

Added together, the single-pixel renderer requires a total of 79 MB. There remains some rooms for optimization: For instance, one channel of the mipmap levels is unused (OpenGL does not offer three-channel images), and for some textures, using 16-bit floats instead of 32-bit float might be sufficient.

5.4.4 Problems with High Geometric Density

A more demanding test case is the San Miguel scene with 7.9M triangles. The most apparent problem of the presented implementation is the lack of adaptivity to the current viewport in addition to the lack of LOD methods. As a result, all 7.9M triangles are used every frame to render the ISMs, with corresponding performance results: The point collection phase of the single-pixel renderer now takes 12.0 ms, while the point rendering takes an additional 5.3 ms. The time needed by the postprocessing is unchanged since it works on fixed resolution textures. The splat renderer takes 12.0 ms, which is the exact same amount of time as the point collection phase of the single-pixel renderer, even though the latter has no rasterization to perform. This demonstrates how this process is bottlenecked by geometric complexity.

Figure 5.7 shows an example screenshot of the San Miguel scene. The reason for the exaggerated shadow is that during ISM rendering, all points are rendered into at least one pixel, which makes them take out too much light if they are actually smaller than one pixel. This is especially the case with the thin structures of the chairs and trees.

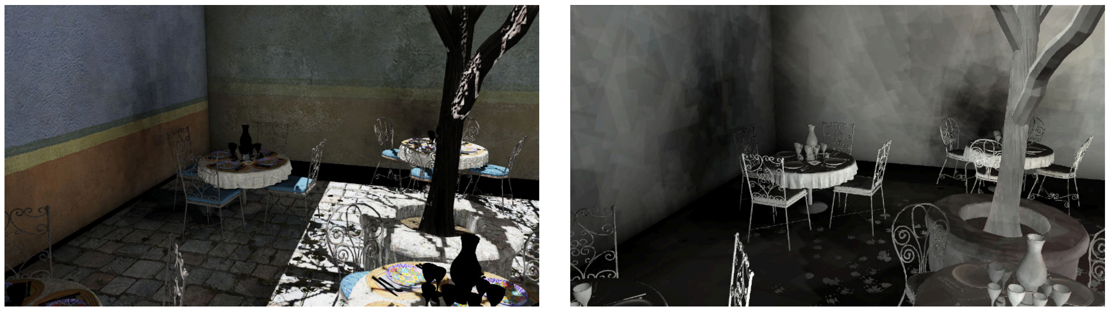

Figure 5.7: *Particularly inaccurate shadows produced by ISMs. The right image shows indirect light only. Ideally, the shadows on the walls would be much smoother. The low-frequency noise is caused by choosing random points per ISM, while the many edges of the shadows are caused by the basic shadow mapping algorithm, which performs a simple binary choice.*

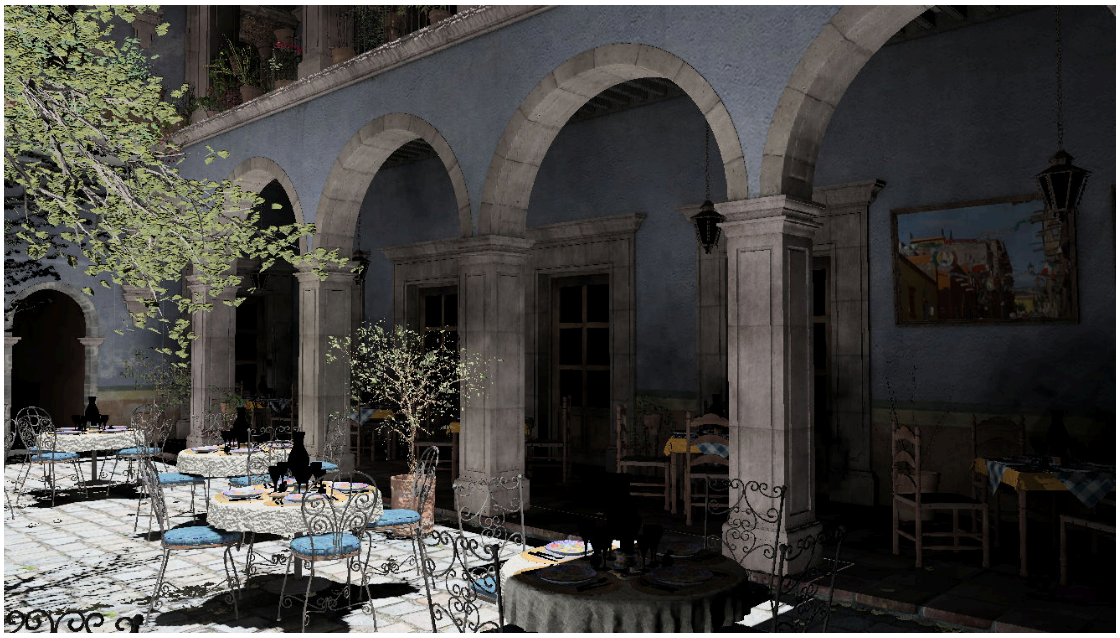

Figure 5.8: *A rendering of the San Miguel scene in the presented implementation. Note the unnaturally dark region behind the pillars.*

The overdarkening is visible in Figure 5.8 as well. Here, the missing second bounce makes the situation even worse: Since most of the VPLs are placed at floor level, the floor behind the pillars is not lit at all.

5.4.5 Comparison of the Splat and Single-Pixel Renderer

The single-pixel renderer provides numerous advantages over the splat renderer: Using a compute shader to render points lifts the restrictions of the fixed-function pipeline and, e. g., allows rendering each point into multiple ISMs without large performance losses. Quality-wise, it can reproduce surfaces more accurately by correctly interpolating between points. However, reproducing the silhouettes of objects accurately is not possible for both renderers; this requires at least using a more complete set of points per ISM, and possibly returning to using triangles instead of points.

The drawbacks obviously include the additional rendering time and memory. Although having moderate requirements in absolute terms, using the single-pixel renderer makes ISM rendering the second most time-consuming and most memory-consuming rendering phase, despite using low resolutions and an approximated scene representation. Besides, the added complexity should not be underestimated. Specifically, we found it hard to fine-tune the pull-push algorithm to produce satisfactory results, especially in the face of the low resolutions of ISMs and the limited data available for reconstruction after selecting a random and sparse set of points.

5.4.6 Discussion

Imperfect shadow maps have several deficits, some of which are inherent to the technique and difficult to solve, while others are specific to the implementation choices and tradeoffs in the presented implementation and might be easier to overcome.

First, ISMs reduce the scene's geometry to points. At the resolution that is achievable in a real-time budget, this is a very rough approximation that loses a lot of accuracy.

Second, because a sparse set of points is used for each ISM, each point must be enlarged to accommodate for the neighboring points that are likely missing in the chosen ISM. For instance, if an area is represented by a thousand points and only one tenth of all points is used for each ISM, each remaining point's area must be enlarged by a factor of ten to result in an equally large area in the rendered output. Of course, this approach leads to deformed geometry since points at the edges of the area are enlarged as well, growing over the borders of the original geometry.

Third, ISMs also have numerical issues. For instance, both the splatting and postprocessing approaches ignore the distortion caused by the paraboloid projection, and thus might render even simple surfaces incorrectly. This contributes to the necessity of using a relatively large shadow bias. At the cost of additional complexity, this can likely be solved with relative ease.

Fourth, and this is in our view the most important shortcoming, ISMs handle geometry in the vicinity to VPLs badly (Figure 5.5). To sufficiently approximate such surfaces when rendering ISMs, one would need to greatly increase the number of points created near VPLs, possibly in addition to stepping away from using a fully random approach for point selection, and one would have to select a specific point set for each ISM depending on the VPLs location.

As for most global illumination algorithms, a massive improvement would be to differentiate between large-scale scene geometry that is important for diffuse light bounces, and smaller geometry of lesser importance. While the latter can possibly be ignored altogether with only minor losses in quality, the shape of large-scale geometry is all the more important to preserve (relatively) precisely.

The San Miguel scene is a good example for this: While the small detail work like dishes and small plants make up most of the geometry and thus most of the rendering time, they contribute only very little to global illumination. This is where the presented implementation suffers most from the lack of level-of-detail methods.

5.5 Interleaved Sampling Using Compute Shaders

Interleaved sampling substantially reduces the number of lights that are processed per pixel. Using a 4x4 interleaving pattern, each pixel processes only a sixteenth of all lights, ideally speeding up the final gathering phase by a factor of 16. However, an additional blur pass is required to mask the noise that results from interleaving, which takes additional time.

Table 5.5: *Timings of interleaved sampling (IS) with a block size of 4x4. Note that the blur pass is mandatory when using interleaved sampling; the "Speedup" column merely illustrates the efficiency of the presented implementation, while the "Speedup with Blur" column shows the practical speedup that is achieved.*

Without IS	With IS	Speedup	Blur Pass	Speedup with Blur
40.24 ms	2.61 ms	15.42	0.63 ms	12.42

Table 5.5 shows timings of the presented implementation. Most notably, the interleaved sampling technique introduces very little overhead, as indicated by the speedup of 15.42 compared to the theoretical maximum of 16. In absolute terms, a speedup of 16 would require the final gathering phase to take 2.51 ms. As a result, the technique introduces 0.1 ms of overhead.

When considering the blur phase, the overhead adds up to 0.73 ms and the speedup is lowered to 12.42. Of course this is still a substantial improvement, especially considering the negligible quality impact.

This technique requires an additional buffer for storing the intermediate result of the blur phase, while the final result can be written into the same texture that was used for final gathering. The texture format `GL_R11F_G11F_B10F` is used here, resulting in roughly 2 MB additional storage at full HD. The presented implementation does not re-use the final gathering texture to keep the results of all stages available for debugging purposes.

Discussion

Interleaved sampling is an obvious win for many-light global illumination methods. Paying the full cost of evaluating all VPLs per shading point is simply unnecessary due to the low-frequency nature of global illumination. Adapting the shading frequency to a similarly low level is the only reasonable choice for real-time applications.

In fact, given the unnoticeable quality impact (Figure 5.9) of interleaved sampling when using a 4x4 block size, larger block sizes, e. g., 8x8 as used by Hedman *et al.* (2016) should be considered. Especially with the advent of high-resolution displays, lowering the shading frequency of global illumination effect even further might be necessary to keep the costs down to a reasonable level. However, the performance gains from lower sample counts will be offset in part by larger blur radii that come with larger block sizes.

Going further, this technique is limited by how well the blur phase is able to mask the noise in more difficult areas with lots of depth discontinuities like vegetation. To this end, one option might be to collect fragments that receive too few information during shading and blurring, and re-process them with larger sets of VPLs in a second pass (similar to Lauritzen, 2010). Temporal reprojection methods (Jimenez, 2016) that re-use information from previous frames and use different sets of VPLs per frame would be another natural extension of interleaved sampling.

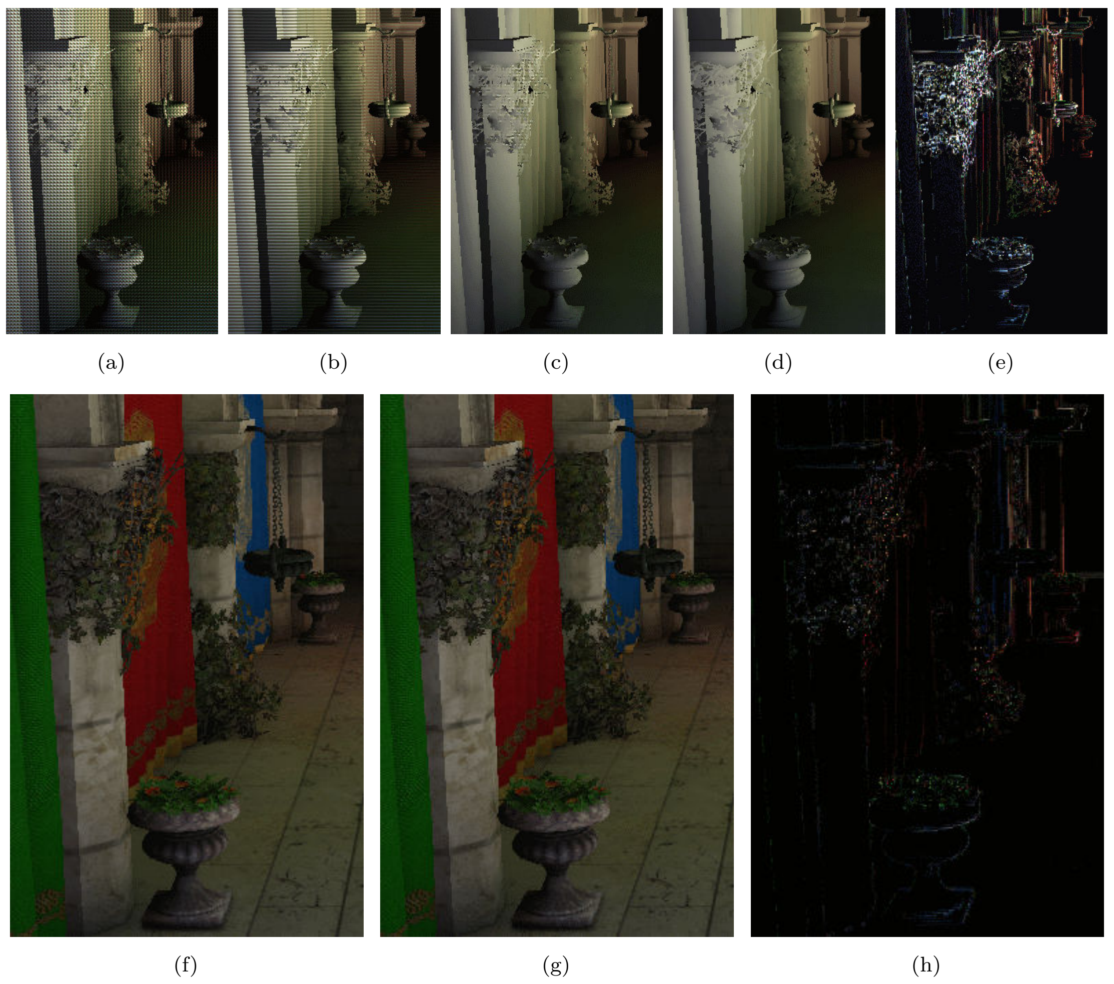

Figure 5.9: *Interleaved sampling results. The top row shows indirect light only. From left to right: (a) Result of the final gathering phase without the bilateral blur, (b) with the horizontal phase of the blur applied, (c) with both phases applied, (d) result of the final gathering phase without using interleaved sampling, (e) difference between (c) and (d). Bottom row: (f) final result after blending when using interleaved sampling, (g) without interleaved sampling, (h) difference between (f) and (g). Difference values are multiplied by eight.*

5.6 Clustered and Tiled Deferred Shading

Clustered deferred shading and tiled deferred shading reduce the number of lighting computations by clustering fragments and culling lights against those clusters. As such, if implemented correctly, they affect only the performance and do not alter the output.

As visible in Table 5.6, both clustered shading and tiled deferred shading can improve the performance of the final gathering phase considerably: In one of the measurements 50 % of the computation time is saved with tiled shading.

It is interesting that clustered shading, being the successor of tiled shading, performs noticeably worse in the context of many-light methods. This is true even for the San Miguel scene, in which the higher amount of depth discontinuities should be favorable

Table 5.6: *Timings of the final gathering stage without optimizations, with clustered shading (CS) and with tiled shading (TS). Each line is a different camera position. Note that the timing "With CS" includes roughly 0.06 ms for the clustering phase and 0.13 ms for the light list phase.*

Scene	Without CS/TS	With CS	With TS
Sponza 1	3.41 ms	2.80 ms	2.40
Sponza 2	3.94 ms	3.63 ms	3.09
Sponza 3	2.68 ms	1.64 ms	1.34
San Miguel	4.75 ms	5.21 ms	4.43

to clustered shading. While these performance characteristics might be specific to the presented implementation, there are plausible reasons that they might apply generally: Due to the infinite light radii used, the percentage of culled lights is generally low. Therefore it might be the better choice to optimize for the worst case and implement a low overhead solution like tiled shading, instead of a more accurate solution like clustered shading that might improve the culling rate, but comes with more overhead.

An impression of the amount of overhead introduced by clustered shading is given by the San Miguel scene, from which the last line of measurements is taken. This scene runs much slower even when only considering the final gathering phase, because most VPLs are placed on the walls and floor, pointing inwards or upwards respectively. Therefore much of the geometry lies on the illuminated side of most VPLs, resulting in very little performance gains from culling.

With a low culling rate, clustered shading actually slows down the final gathering phase by 0.46 ms. While 0.19 ms of these are fixed costs added by the clustering and light list calculation phases, the final gathering shader was slowed down by 0.27 ms as well. This might be caused by the indirection when accessing lights, since instead of reading from the light buffer directly, the light list with indices is accessed first before reading the light buffer with a dynamic index. Another explanation might be the divergent data flow, since threads from the same work group can access different light lists and thus different light data and shadow maps, possibly hurting cache efficiency.

As discussed in section Section 4.6, clustered shading consumes about 2 MB of memory. The tiled shading algorithm however, being integrated into the final gathering shader, uses no additional VRAM at all.

Discussion

The presented measurements make tiled shading seem like a clear winner, but as so often the situation is more complicated. The integration of tiled shading into the final gathering shader, while an efficient and easy to implement solution, is inherently less flexible than clustered shading. For instance, Olsson *et al.* (2012) propose to use normals as another dimension besides the three spatial ones, further improving culling efficiency. While this is presumably easy to integrate into the more flexible approach taken with clustered shading, the integrated tiled shading is constrained by the amount of available shared memory, which makes it challenging to take additional dimensions into account.

Chapter 6

Conclusion and Future Work

In this thesis we have presented a rendering system that simulates global illumination at real-time framerates. It takes only a few milliseconds for computing indirect lighting, making it suitable for use in real-time applications on today's hardware.

In addition to imperfect shadow maps rendered with splatting, a second renderer based on compute shaders has been implemented. With the flexibility provided by compute shaders, it is able to render a high amount of points more efficiently compared to the splat renderer, which relies on geometry shaders. Extended with a high-quality post-processing, it renders shadow maps of noticeable higher quality than the splat renderer, but the resulting quality is limited by the input data, a randomly selected and sparse set of points representing the scene. Several failure cases have been identified where the presented implementation generates artifacts clearly visible even to the untrained eye.

Interleaved shading has traditionally been performed by the means of several shader passes, which split a buffer into several smaller ones, perform processing on them, and merge these buffers again. This thesis has presented an implementation that is integrated into the final gathering shader, making splitting and merging buffers obsolete while still processing the workload cache-coherently. Its near-perfect efficiency, ease of implementation, and the fact that it requires no additional memory make this implementation the obvious choice when compute shaders are available on the given platform.

Two light culling techniques well-known in the video games industry, tiled and clustered deferred shading, have been applied to many-light global illumination. While tiled deferred shading's simplicity allowed for a compact implementation integrated into the final gathering shader, clustered deferred shading promised higher culling efficiency at the expense of higher overhead. The results have shown this overhead to be not worth the gains, making clustered deferred shading not the perfect candidate for light culling in a global illumination context using many-light methods. There are extensions, however, that could improve its culling efficiency to the point where it causes no slowdown in the worst case. Tiled deferred shading, on the other hand, is a clear performance win and again, rather easy to implement, but limited in its maximum culling efficiency.

Looking forward, going beyond random point sampling for creating imperfect shadow maps is desirable, since the randomness is limiting their quality. The implementation of more advanced surface reconstruction techniques, starting with the full extent of the algorithm presented by Marroquim *et al.* (2007), would likely further enhance the ISM's quality.

Given how common the usage of interleaved sampling is, it is surprising that little work has been done to maximize the technique's impact. For instance, thoroughly investigating the correlation between performance and quality when changing the block size would give implementers a guideline on which block sizes to choose in which scenarios.

There is still potential in the clustered deferred shading technique, for instance by using surface normals for more efficient culling. Mipmapping the ISMs and using them for culling, performing an early shadow lookup for entire clusters of fragments, would be interesting as well.

However, the final gathering phase is quite fast at least on high-end hardware, and, more importantly, not detrimental to quality. While rendering the ISMs is fast as well, their quality is lacking and limiting the potential use of the presented technique. Improving their precision by, for example, improving the point sampling or surface reconstruction or investigating triangle-based approaches, is in our view the path with the largest potential benefit.

Bibliography

Abrash, Michael (1997). *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash*. 10th. Coriolis Group Books. ISBN: 1576101746.

Andersson, Johan (2011). *DirectX 11 rendering in Battlefield 3*. Presentation at GDC 2011. URL: http://www.dice.se/news/directx-11-rendering-battlefield-3/.

Barák, Tomá, Jirí Bittner, and Vlastimil Havran (2013). "Temporally coherent adaptive sampling for imperfect shadow maps". In: *Computer Graphics Forum*. Vol. 32. 4. Wiley Online Library, pp. 87–96.

Bunnell, Michael (2005). "Dynamic Ambient Occlusion and Indirect Lighting". In: *GPU Gems 2*. Ed. by Matt Pharr. Addison-Wesley, pp. 223–233.

Chen, Y.-Y., Y.-J. Chen, and S.-Y. Chien (2016). "Quantizing Intersections Using Compact Voxels". In: *Computer Graphics Forum*. ISSN: 1467-8659. URL: http://dx.doi.org/10.1111/cgf.12855.

Christensen, P (2008). "Point-based approximate color bleeding". In: *Pixar Technical Notes* 2.5, p. 6. URL: http://graphics.pixar.com/library/PointBasedColorBleeding/paper.pdf.

Cigolle, Zina H., Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer (2014). "A Survey of Efficient Representations for Independent Unit Vectors". In: *Journal of Computer Graphics Techniques (JCGT)* 3.2, pp. 1–30. ISSN: 2331-7418. URL: http://jcgt.org/published/0003/02/01/.

Crassin, Cyril and Simon Green (2012). "Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer". In: *OpenGL Insights*. URL: http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf (Chapter PDF), http://blog.icare3d.org/2012/05/gtc-2012-talk-octree-based-sparse.html (GTC 2012 Talk). CRC Press, Patrick Cozzi and Christophe Riccio.

Dachsbacher, Carsten, Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Bruce Walter, and Jan Novák (2014). "Scalable Realistic Rendering with Many-Light Methods". In: *Computer Graphics Forum* 33.1, pp. 88–104. ISSN: 1467-8659.

Dachsbacher, Carsten and Marc Stamminger (2005). "Reflective Shadow Maps". In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. ACM, pp. 203–231. ISBN: 1-59593-013-2. URL: http://doi.acm.org/10.1145/1053427.1053460.

Dachsbacher, Carsten and Marc Stamminger (2006). “Splatting Indirect Illumination”. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D ’06. ACM, pp. 93–100. ISBN: 1-59593-295-X. URL: http://doi.acm.org/10.1145/1111411.1111428.

Dong, Zhao, Thorsten Grosch, Tobias Ritschel, Jan Kautz, and Hans-Peter Seidel (2009). “Real-time Indirect Illumination with Clustered Visibility.” In: *VMV*, pp. 187–196.

Donnelly, William and Andrew Lauritzen (2006). “Variance Shadow Maps”. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D ’06. ACM, pp. 161–165. ISBN: 1-59593-295-X. URL: http://doi.acm.org/10.1145/1111411.1111440.

Faure, William and Chun-Fa Chang (2010). “Metalights: Improved Interleaved Shading”. In: *Computer Graphics Forum* 29.7, pp. 2109–2117. ISSN: 1467-8659. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01798.x.

Georgiev, Iliyan and Philipp Slusallek (2010). “Simple and robust iterative importance sampling of virtual point lights”. In: *Proceedings of Eurographics (short papers)* 4.

Günther, Christian, Thomas Kanzok, Lars Linsen, and Paul Rosenthal (2013). “A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds”. In: *Journal of WSCG* 21.2. Ed. by Vaclav Skala, pp. 153–161. ISSN: 1213-6972. URL: http://www.paul-rosenthal.de/wp-content/uploads/2013/06/guenther-wscg-2013.pdf.

Harris, Mark (2007). *Optimizing parallel reduction in CUDA*. NVIDIA Developer Technology. URL: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

Hedman, Peter, Tero Karras, and Jaakko Lehtinen (2016). “Sequential Monte Carlo Instant Radiosity”. In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’16. ACM, pp. 121–128. ISBN: 978-1-4503-4043-4. URL: http://doi.acm.org/10.1145/2856400.2856406.

Jensen, Henrik Wann (1996). “Global Illumination Using Photon Maps”. In: *Proceedings of the Eurographics Workshop on Rendering Techniques ’96*. Springer-Verlag, pp. 21–30. ISBN: 3-211-82883-4. URL: http://dl.acm.org/citation.cfm?id=275458.275461.

Jimenez, Jorge (2016). *Filmic SMAA: Sharp Morphological and Temporal Antialiasing*. Presentation at SIGGRAPH 2016. URL: http://advances.realtimerendering.com/s2016/index.html.

Jiménez, Jorge (2016). *Practical Real-Time Strategies for Accurate Indirect Occlusion*. Presentation at SIGGRAPH 2016. URL: http://blog.selfshadow.com/publications/s2016-shading-course/activision/s2016_pbs_activision_occlusion.pdf.

Kajiya, James T. (1986). “The Rendering Equation”. In: *SIGGRAPH Comput. Graph.* 20.4, pp. 143–150. ISSN: 0097-8930. URL: http://doi.acm.org/10.1145/15886.15902.

Kaplanyan, Anton (2009). “Light propagation volumes in cryengine 3”. In: *ACM SIGGRAPH Courses* 7, p. 2. URL: http://www.crytek.com/download/Light_Propagation_Volumes.pdf.

Kaplanyan, Anton and Carsten Dachsbacher (2010). “Cascaded Light Propagation Volumes for Real-time Indirect Illumination”. In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’10. ACM, pp. 99–107. ISBN: 978-1-60558-939-8. URL: http://doi.acm.org/10.1145/1730804.1730821.

Keller, Alexander (1997). “Instant Radiosity”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97. ACM Press/Addison-Wesley Publishing Co., pp. 49–56. ISBN: 0-89791-896-7. URL: http://dx.doi.org/10.1145/258734.258769.

Keller, Alexander and Wolfgang Heidrich (2001). “Interleaved Sampling”. In: *Rendering Techniques 2001: Proceedings of the Eurographics Workshop in London, United Kingdom, June 25–27, 2001*. Ed. by Steven J. Gortler and Karol Myszkowski. Springer Vienna, pp. 269–276. ISBN: 978-3-7091-6242-2. URL: http://dx.doi.org/10.1007/978-3-7091-6242-2_25.

Krivanek, Jaroslav, Pascal Gautron, Sumanta Pattanaik, and Kadi Bouatouch (2005). “Radiance caching for efficient global illumination computation”. In: *IEEE Transactions on Visualization and Computer Graphics* 11.5, pp. 550–561.

Laine, Samuli and Tero Karras (2011). “High-performance Software Rasterization on GPUs”. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG ’11. ACM, pp. 79–88. ISBN: 978-1-4503-0896-0. URL: http://doi.acm.org/10.1145/2018323.2018337.

Laine, Samuli, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila (2007). “Incremental Instant Radiosity for Real-time Indirect Illumination”. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR’07. Eurographics Association, pp. 277–286. ISBN: 978-3-905673-52-4. URL: http://dx.doi.org/10.2312/EGWR/EGSR07/277-286.

Lauritzen, Andrew (2010). *Deferred rendering for current and future rendering pipelines*. Presentation at SIGGRAPH 2010. URL: http://bps10.idav.ucdavis.edu/.

Mara, Michael, Morgan McGuire, Derek Nowrouzezahrai, and David Luebke (2014). *Fast Global Illumination Approximations on Deep G-Buffers*. Tech. rep. NVR-2014-001. NVIDIA Corporation, p. 16.

— (2016). “Deep G-buffers for Stable Global Illumination Approximation”. In: *Proceedings of High Performance Graphics*. HPG ’16. Eurographics Association, pp. 87–98. ISBN: 978-3-03868-008-6. URL: http://dx.doi.org/10.2312/hpg.20161195.

Marroquim, Ricardo, Martin Kraus, and Paulo Roma Cavalcanti (2007). “Efficient Point-Based Rendering Using Image Reconstruction”. In: *Proceedings Symposium on Point-Based Graphics*, pp. 101–108.

— (2008). “Special Section: Point-Based Graphics: Efficient Image Reconstruction for Point-based and Line-based Rendering”. In: *Comput. Graph.* 32.2, pp. 189–203. ISSN: 0097-8493. URL: http://dx.doi.org/10.1016/j.cag.2008.01.011.

McGuire, Morgan (2011). *Computer Graphics Archive.* URL: http://graphics.cs.williams.edu/data.

Mittring, Martin (2007). "Finding Next Gen: CryEngine 2". In: *ACM SIGGRAPH 2007 Courses.* SIGGRAPH '07. ACM, pp. 97–121. ISBN: 978-1-4503-1823-5. URL: http://doi.acm.org/10.1145/1281500.1281671.

Nichols, Greg and Chris Wyman (2009b). "Multiresolution Splatting for Indirect Illumination". In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games.* I3D '09. ACM, pp. 83–90. ISBN: 978-1-60558-429-4. URL: http://doi.acm.org/10.1145/1507149.1507162.

Novák, Jan, Thomas Engelhardt, and Carsten Dachsbacher (2011). "Screen-space bias compensation for interactive high-quality global illumination with virtual point lights". In: *Symposium on Interactive 3D Graphics and Games.* ACM, pp. 119–124.

Olsson, Ola and Ulf Assarsson (2011). "Tiled Shading". In: *Journal of Graphics, GPU, and Game Tools* 15.4, pp. 235–251. URL: http://www.tandfonline.com/doi/abs/10.1080/2151237X.2011.621761.

Olsson, Ola, Markus Billeter, and Ulf Assarsson (2012). "Clustered Deferred and Forward Shading". In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics.* EGGH-HPG'12. Eurographics Association, pp. 87–96. ISBN: 978-3-905674-41-5. URL: http://dx.doi.org/10.2312/EGGH/HPG12/087-096.

Olsson, Ola, Emil Persson, and Markus Billeter (2015). "Real-time Many-light Management and Shadows with Clustered Shading". In: *ACM SIGGRAPH 2015 Courses.* SIGGRAPH '15. ACM, 12:1–12:398. ISBN: 978-1-4503-3634-5. URL: http://doi.acm.org/10.1145/2776880.2792712.

Panteleev, Alexey (2015). *NVIDIA VXGI: Dynamic Global Illumination for Games.* Presentation at GPU Technology Conference 2015. URL: http://on-demand.gputechconf.com/gtc/2015/presentation/S5670-Alexey-Panteleev.pdf.

Persson, Emil and Ola Olsson (2013). *Practical Clustered Deferred and Forward Shading.* SIGGRAPH Course: Advances in Real-Time Rendering in Games. URL: http://s2013.siggraph.org/attendees/courses/session/advances-real-time-rendering-games-part-i.

Prutkin, Roman, Anton Kaplanyan, and Carsten Dachsbacher (2012). "Reflective Shadow Map Clustering for Real-Time Global Illumination." In: *Eurographics (Short Papers)*, pp. 9–12.

Reed, Nathan (2012). *Ambient Occlusion Fields and Decals in Infamous 2.* Presentation at GDC 2012. URL: http://www.gdcvault.com/play/1015320/Ambient-Occlusion-Fields-and-Decals.

Rehfeld, Hauke, Tobias Zirr, and Carsten Dachsbacher (2014). "Clustered Pre-convolved Radiance Caching". In: *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization.* PGV '14. Eurographics Association, pp. 25–32. ISBN: 978-3-905674-59-0. URL: http://dx.doi.org/10.2312/pgv.20141081.

Ritschel, Tobias, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz (2012). "The State of the Art in Interactive Global Illumination". In: *Comput. Graph. Forum* 31.1, pp. 160–188. ISSN: 0167-7055. URL: http://dx.doi.org/10.1111/j.1467-8659.2012.02093.x.

Ritschel, Tobias, Elmar Eisemann, Inwoo Ha, James D. K. Kim, and Hans-Peter Seidel (2011). "Making Imperfect Shadow Maps View-Adaptive: High-Quality Global Illumination in Large Dynamic Scenes". In: *Computer Graphics Forum* 30.8, pp. 2258–2269. ISSN: 1467-8659. URL: http://dx.doi.org/10.1111/j.1467-8659.2011.01998.x.

Ritschel, Tobias, Thomas Engelhardt, Thorsten Grosch, Hans-Peter Seidel, Jan Kautz, and Carsten Dachsbacher (2009a). "Micro-rendering for Scalable, Parallel Final Gathering". In: *ACM Trans. Graph.* 28.5, 132:1–132:8. ISSN: 0730-0301. URL: http://doi.acm.org/10.1145/1618452.1618478.

Ritschel, Tobias, Thorsten Grosch, Min H Kim, H-P Seidel, Carsten Dachsbacher, and Jan Kautz (2008). "Imperfect shadow maps for efficient computation of indirect illumination". In: *ACM Transactions on Graphics (TOG)* 27.5, p. 129.

Ritschel, Tobias, Thorsten Grosch, and Hans-Peter Seidel (2009b). "Approximating Dynamic Global Illumination in Image Space". In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. ACM, pp. 75–82. ISBN: 978-1-60558-429-4. URL: http://doi.acm.org/10.1145/1507149.1507161.

Saito, Takafumi and Tokiichiro Takahashi (1990). "Comprehensible Rendering of 3-D Shapes". In: *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '90. ACM, pp. 197–206. ISBN: 0-89791-344-2. URL: http://doi.acm.org/10.1145/97879.97901.

Scherzer, Daniel, Chuong H Nguyen, Tobias Ritschel, and Hans-Peter Seidel (2012). "Pre-convolved Radiance Caching". In: *Computer Graphics Forum*. Vol. 31. 4. Wiley Online Library, pp. 1391–1397.

Segovia, Benjamin, Jean Claude Iehl, Richard Mitanchey, and Bernard Péroche (2006a). "Bidirectional Instant Radiosity." In: *Rendering Techniques*, pp. 389–397.

— (2006b). "Non-interleaved deferred shading of interleaved sample patterns". In: *SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware: Proceedings of the 21 st ACM SIGGRAPH/Eurographics symposium on Graphics hardware: Vienna, Austria*. Vol. 3. 04, pp. 53–60.

Sloan, Peter-Pike, Jan Kautz, and John Snyder (2002). "Precomputed Radiance Transfer for Real-time Rendering in Dynamic, Low-frequency Lighting Environments". In: *ACM Trans. Graph.* 21.3, pp. 527–536. ISSN: 0730-0301. URL: http://doi.acm.org/10.1145/566654.566612.

Stefanov, Nikolay (2012). *Deferred Radiance Transfer Volumes: Global Illumination in Far Cry 3*. Presentation at GDC 2012. URL: http://www.gdcvault.com/play/1015326/Deferred-Radiance-Transfer-Volumes-Global.

Strengert, Magnus, Martin Kraus, and Thomas Ertl (2006). "Pyramid Methods in GPU-Based Image Processing". In: *Workshop on Vision, Modelling, and Visualization VMV '06*, pp. 169–176.

Sun, Che and Emmanuel Agu (2015). “Advances in Visual Computing: 11th International Symposium, ISVC 2015, Las Vegas, NV, USA, December 14-16, 2015, Proceedings, Part II”. In: ed. by George Bebis *et al.* Springer International Publishing. Chap. Many-Lights Real Time Global Illumination Using Sparse Voxel Octree, pp. 150–159. ISBN: 978-3-319-27863-6. URL: http://dx.doi.org/10.1007/978-3-319-27863-6_14.

Thiedemann, Sinje, Niklas Henrich, Thorsten Grosch, and Stefan Müller (2011). “Voxel-based Global Illumination”. In: *Symposium on Interactive 3D Graphics and Games*. I3D ’11. ACM, pp. 103–110. ISBN: 978-1-4503-0565-5. URL: http://doi.acm.org/10.1145/1944745.1944763.

Tokuyoshi, Yusuke (2015). “Virtual Spherical Gaussian Lights for Real-time Glossy Indirect Illumination”. In: *Computer Graphics Forum* 34.7, pp. 89–98. ISSN: 1467-8659. URL: http://dx.doi.org/10.1111/cgf.12748.

Tokuyoshi, Yusuke and Takahiro Harada (2016). “Stochastic Light Culling”. In: *Journal of Computer Graphics Techniques (JCGT)* 5.1, pp. 35–60. ISSN: 2331-7418. URL: http://jcgt.org/published/0005/01/02/.

Tokuyoshi, Yusuke and Shinji Ogaki (2012). “Real-time Bidirectional Path Tracing via Rasterization”. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’12. ACM, pp. 183–190. ISBN: 978-1-4503-1194-6. URL: http://doi.acm.org/10.1145/2159616.2159647.

Ward, Gregory J., Francis M. Rubinstein, and Robert D. Clear (1988). “A Ray Tracing Solution for Diffuse Interreflection”. In: *SIGGRAPH Comput. Graph.* 22.4, pp. 85–92. ISSN: 0097-8930. URL: http://doi.acm.org/10.1145/378456.378490.

Further Reading

Chang, Byungjoon, Sanghun Park, and Insung Ihm (2015). “Ray tracing-based interactive diffuse indirect illumination”. In: *Multimedia Tools and Applications*.

Evans, Alex (2015). “Learning from Failure: a Survey of Promising, Unconventional and Mostly Abandoned Renderers for “Dreams PS4”, a Geometrically Dense, Painterly UGC Game”. SIGGRAPH 2015. URL: http://advances.realtimerendering.com/s2015/.

Harada, Takahiro (2014). “Micro-buffer Rasterization Reduction Method for Environment Lighting Using Point-based Rendering”. In: *Proceedings of Graphics Interface 2014*. GI ’14. Canadian Information Processing Society, pp. 95–102. ISBN: 978-1-4822-6003-8. URL: http://dl.acm.org/citation.cfm?id=2619648.2619664.

Hollander, Matthias, Tobias Ritschel, Elmar Eisemann, and Tamy Boubekeur (2011). “ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination”. In: *Computer Graphics Forum*. Vol. 30. 4. Wiley Online Library, pp. 1233–1240.

Klehm, Oliver (2010). “Interactive Massive Lighting for Virtual 3D City Models”. MA thesis. Hasso Plattner Institute, University of Potsdam.

Knecht, Martin (2009). “Real-Time Global Illumination Using Temporal Coherence”. MA thesis. Institute of Computer Graphics and Algorithms, Vienna University of Technology. URL: https://www.cg.tuwien.ac.at/research/publications/2009/knecht-2009-MKN/.

Křivánek, Jaroslav, Miloš Hašan, Adam Arbree, Carsten Dachsbacher, Alexander Keller, and Bruce Walter (2012). “Optimizing realistic rendering with many-light methods”. In: *ACM SIGGRAPH 2012 Courses*. SIGGRAPH ’12. ACM, 7:1–7:217. ISBN: 978-1-4503-1678-1. URL: http://doi.acm.org/10.1145/2343483.2343490.

Laurent, Gilles, Cyril Delalandre, Gregoire de La Riviere, and Tamy Boubekeur (2016). “Forward Light Cuts: A Scalable Approach to Real-Time Global Illumination”. In: *Compurer Graphics Forum (Proc. EGSR 2016)* 35.4, pp. 79–88.

Nichols, Greg, Jeremy Shopf, and Chris Wyman (2009a). “Hierarchical Image-Space Radiosity for Interactive Global Illumination”. In: *Computer Graphics Forum*. Vol. 28. 4. Wiley Online Library, pp. 1141–1149.

Noël, Laurent and Venceslas Biri (2014). “Real-Time Global Illumination for Games using Topological Information”. In: *International Conference on Computer Games, Multimedia & Allied Technology (CGAT). Proceedings*. Global Science and Technology Forum, p. 65.

Olsson, Ola, Erik Sintorn, Viktor Kämpe, Markus Billeter, and Ulf Assarsson (2014). "Efficient Virtual Shadow Maps for Many Lights". In: *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '14. ACM, pp. 87–96. ISBN: 978-1-4503-2717-6. URL: http://doi.acm.org/10.1145/2556700.2556701.

Sugihara, Masamichi, Randall Rauwendaal, and Marco Salvi (2014). "Layered Reflective Shadow Maps for Voxel-based Indirect Illumination." In: *High Performance Graphics*, pp. 117–125.

Wiesenhütter, Daniel, Andreas Klein, and Alfred Nischwitz (2013). "LightCluster: clustering lights to accelerate shadow computation". In: *ACM SIGGRAPH 2013 Posters*.

Yu, Insu, Andrew Cox, Min H. Kim, Tobias Ritschel, Thorsten Grosch, Carsten Dachsbacher, and Jan Kautz (2009). "Perceptual Influence of Approximate Visibility in Indirect Illumination". In: *ACM Transactions on Applied Perception (TAP)* 6.4, 24:1–24:14. ISSN: 1544-3558. URL: http://doi.acm.org/10.1145/1609967.1609971.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Potsdam, December 23, 2016	
(Ort, Datum)	(Unterschrift)