INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Incremental Light Propagation Volumes In CAVE-Like Installations

Benjamin Sommer

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Incremental Light Propagation Volumes In CAVE-Like Installations

Benjamin Sommer

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Christoph Anthes
Abgabetermin:	24. September 2014

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

München, den 2. September 2014

. . . . . .
(Benjamin Sommer)

Abstract

This thesis presents a new application of Light Propagation Volumes (LPVs) to multi-display environments and improves upon previous work. It introduces the novel method Incremental Light Propagation Volumes (ILPVs) to reuse propagated light from previous frames to increase propagation distance which significantly enhances rendering quality at almost no additional cost. Transfer vectors and the geometry lattice are computed with stochastic sampling. Energy conservation during each propagation iteration is approximated by stochastically sampling uniform scale factors for all transfer vectors. During propagation, light can be absorbed and scattered. Both absorption and scattering coefficients are normalized by taking mean neighbor cell distance into account. Initialization including precomputations requires about 10 seconds, depending on scene complexity. During runtime, the multithreaded synchronized application using GPGPU delivers 30 to 60 Hz in a multi-display installation.

Diese Bachelorarbeit präsentiert eine neue Anwendung von LPVs in Multi-Display Umgebungen und verbessert bisherige Arbeiten. Die neue Methode ILPV wird eingeführt welche propagiertes Licht von vorherigen Berechnungen wiederverwendet, um dadurch die bisher sehr limitierte Propagierungsdistanz signifikant zu vergrößern bei gleichzeitig sehr geringen zusätzlichen Rechenaufwand. Transfer Vektoren sowie das Geometriegitter werden durch stochastische Samplingverfahren vorberechnet. Energieerhaltung während jeder Propagierungsiteration ist durch stochastisches uniformes Sampling von Skalierungsfaktoren für alle Transfer Vektoren approximiert. Während der Propagierung kann Light absorbiert und in andere Raumrichtungen gebrochen werden. Hier werden die Koeffizienten für Absorption und Brechung mittels mittlerer Distanz zur Nachbarzelle normalisiert. Initialisierung mit Vorberechnungen benötigt ca. 10 Sekunden, abhängig von der Szenenkomplexität. Während der Laufzeit sorgt die synchronisierte Anwendung mit GPGPU für 30 bis 60 Hz in einer Multi-Display Umgebung.

Contents

1 Introduction

> In 2006 Dutré et al. state: "Photorealistic rendering and global illumination algorithms have come a long way since the publication of the first recursive ray-tracing algorithm in 1979. There has been a gradual evolution from simple algorithms [...] to very advanced, fully physically based rendering algorithms. It is now possible, within a reasonable amount of time, to generate an image that is indistinguishable from a photograph of a real scene. [...] As such, photorealistic rendering has certainly propelled forward the development of high-quality visualization techniques." ([DBB06, p. 301])

In CAVE Automatic Virtual Environment (CAVE)[CSD93][1] displays, rendering plausible dynamic indirect illumination is important to keep immersion high. Being a delicate objective, high resolution stereoscopic rendering in high frame rates in a distributed cluster imposes tremendous demands on both the hardware and the algorithms. In particular temporal lag and flickering have to be kept at minimum, while avoiding both jittering and temporal incoherence is crucial. Otherwise immersion breaks and the observer may suffer from cybersickness symptoms, e.g. headache, stomach ache, ataxia, nausea or vomitting. In the situation of stereoscopic rendering, screen-space algorithms may not be the best choice. Instead computations of low-frequent indirect illumination for surfaces seen through left and right camera eye should be shared between eyes as much as possible. This approach is particularly effective for distant surfaces. However, the above objectives must not be penalized.

Multi-display environments help the archeologist to explore and investigate digitalized tombs with a flashlight. The architect may prematurely verify perceived illumination in rooms and buildings during the development phase, experiencing a realistic impression of the new architecture. Intense immersion further enables to work therapeutically with e.g. achluophobia, acrophobia or claustrophobia.

1.1 Motivation

Rendering real time dynamic Global Illumination (GI) in CAVEs is still challenging. Besides Virtual Reality (VR) applications require both navigation and non-trival interaction demanding substantial time to implement properly as well. As such, non-physically based local illumination implementations are widespread in VR applications (figure 1.1).

Recently [KD10] proposed scene-space based, temporally coherent LPV which fits well to stereoscopic rendering, finally corresponding to real time refresh rates. The presented approach in this thesis for computing GI is based on LPV since it additionally avoids flickering, jittering and incoherence. Extensive literature review revealed that LPV has not yet been applied to CAVE installations.

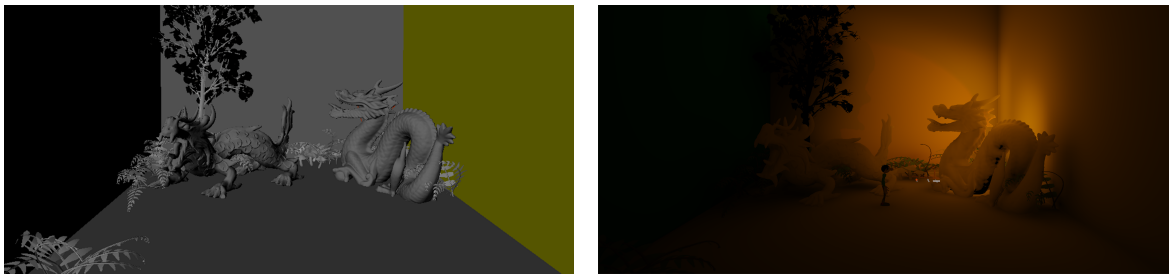
[1] CAVE™ is a registered trademark of the University of Illinois' Board of Trustees. The term is used in the context of this thesis to generically refer to CAVEs™ and CAVE-like displays.

However previous LPV based approaches are inherintly limited by propagation distance of light in the virtual scene. This distance depends on the resolutions of internal data structures to compute indirect illumination. The computational time to distribute light still scales cubically with respect to these resolutions. For high resolutions, propagating light far distances may not even be achievable in real time or the quality of rendered images suffer significantly. To mitigate this limitation, the presented approach in this thesis improves upon previous work by reusing propagated light from previous frames, effectively increasing propagation distance. The changes are straightforward to implement which impose almost no additional computational cost.

Propagating light in the virtual scene is realized by successive local light energy exchanges between structure elements. The latter process requires well defined transfer vectors. This thesis employs stochastic rejection sampling to generate these vectors. In physically based rendering, the algorithms are bound to comply with energy conservation. As such, these transfer vectors need to be scaled to adhere to this law. Although the academic literature in this field note the need for scaling, no further details concerning the underlying algorithms are defined. Therefore this thesis employs a stochastic sampling approach by probing energy differences for both sampled scaling factors and stochastic local light distributions, effectively introducting a minimization problem.

Multi-display installations pose an additional challenge to consistently render indirect shadows between display projections, particularly between neighboring displays to avoid disturbing shadow dislocations in terms of both position and light spectrum. Although previous LPV based approaches propagate light in a camera view independent manner, an additional data structure helping to consider indirect shadows still depends significantly on camera projections resulting in incoherence between displays. Consequently this thesis employs a camera view independent approach to generate this data structure approximating all scene geometry surfaces without holes. This thesis finally shows that illumination is consistently rendered between displays.

In the literature, the propagation stage to distribute light in the virtual scene is presented to be implemented in various frameworks at which each academic publication employs a single framework only, making performance comparisons harder. At the time of writing, extensive literature review revieled the need for a comparison as to which framework works



(a) Non-physically based local illumination (b) Physically based global illumination

Figure 1.1: Comparison between local and global illumination. Non-physically based local illumination approximates direct illumination, possibly without direct shadows. Indirect illumination is traditionally approximated with a constant ambient color. Physically based global illumination computes indirect illumination based on current scene surfaces and scene lighting, resulting in superior image quality, and direct illumination is based on microfacet theory.

most efficient concerning the propagation stage, including technical explanations for either performance gains or performance drops. This thesis additionally presents such a comparison between OpenGL, OpenCL and CUDA whereby the latter proved to be the most efficient framework here.

1.2 Outline

Beginning with a theoretical background, this thesis briefly introduces important radiometric quantities and equations of transfer on which the presented GI rendering approach is based, both of which constitute key insights into how physically based GI renderings may generally be computed, although more knowledge is required than presented here. Subsequently related work is discussed concerning GI in multi-display environments, at which the refresh rate of presented algorithms range from interactive to real time consuming either short or long time for precompution. This section explains as well which approaches fit well to multi-display installations and why a LPV based approach has been chosen in this thesis.

Based on this theoretical background, the next chapter 3 presents the rendering pipeline concept from a general perspective having an apparent emphasis on mathematical formula and theoretical concepts without implementation details. It brings together distinct algorithms to finally constitute the renderer. After introducing the CAVE cluster, this chapter explains initial procedures to properly precompute the geometry lattice as well as complex constants during rendering, e.g. transfer vectors and outscattering fractions. Afterwards focus is given on the runtime behavior, e.g. to inject primary light into the scene lattices, to propagate illumination between lattice cells, to compute both indirect and direct illumination and finally to correctly send scene radiance to the display adapter by employing tone mapping and gamma correction. This chapter presents experiments with surface lights as well. Finally the novel incremental approach to reuse light distributions from previous frames is explained being accompanied with an evaluation of differences to previous approaches in this field.

The next chapter 4 provides implementation details. After introducing the implemented software architecture, propagation in different frameworks is discussed, particularly concerning CUDA giving technical details into writing the highly optimized CUDA kernel. Here performance gains and performance drops are discusses in the frameworks OpenGL and OpenCL as well, although their kernel is not discussed in detail. The final section discusses implemented performance measurement techniques being crucial for both reliable temporal measurements and conclusions based on these data. It compares runtime performance for multiple configurations and for each chosen framework.

The last chapter 5 finally draws conclusions concerning the presented approaches in this thesis. It presents future work to improve upon both this work and LPV based approaches in general which may significantly improve performance or quality of ILPV.

2 Background and Related Work

Beginning with a short theoretical overview concerning radiometry and rendering equations, this chapter primarily summarizes important previous academic work in real time rendering.

Besides explaining publications made in the context of CAVE applications, this chapter explains as well which work has been taken into account, but has been omitted in the presented approach.

2.1 Background Theory

Global illumination algorithms compute the steady-state distribution of light energy in a scene. In this context, the presented approach employs concepts from radiometry theory being the area of study concerning the physical measurement of light. It differs from photometry in which it does not take perception of light energy into account. These radiometric quantities are employed in formal equations of light transfer to compute light distribution in a scene. ILPV is based on the indirect illumination components of these equations.

2.1.1 Radiometry of Light Transport

In radiometry theory[Cha60], radiant power or flux Φ is the amount of energy that flows through a surface per unit time, expressed in watts (J/s). This quantity neither includes the size of light source or receiver, nor the distance betweem them. This size is however taken into account by irradiance E, the incident radiant power on a surface per unit surface area:

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A} \qquad \left[\frac{W}{m^2}\right]$$

Similar to irradiance, radiant exitance M or radiosity B is the exitant radiant power per unit surface area. To further include distances, the very important radiance $L(x, \omega)$ captures the amount of power arriving at a surface point, per unit solid angle and per unit projected area. This quantity varies with position and direction and essentially captures the appearance of objects. An important property is that response of sensors, e.g. camera or eye, is proportional to incident radiance, so GI algorithms must compute radiance [DBB06]. Note that radiance at surface points is sometimes referred to by surface radiance. In addition, computing GI requires a spatial, directional and spectral intensity distribution. In the context of real time algorithms, spectral intensity is often approximated by using three color channels. Concerning diffuse emitters, radiance can be computed by using [DBB06]

$$\Phi = LA\pi = BA.$$

The solid angle Ω measures the size of an area projected onto the unit sphere. Thus the solid angle of a hemisphere equals 2π. Instead of expensively integrating an area, the solid angle

can be approximated for small surfaces A_y by

$$\Omega = \frac{A \cos \alpha}{d^2} = \frac{A_y \cdot \langle \boldsymbol{n_y} | \boldsymbol{r_{xy}} \rangle}{\| \boldsymbol{r_{xy}} \|} \qquad [sr]$$

as seen from point x.

2.1.2 Equations of Transfer

A formalized way to compare and evaluate GI approaches is the Rendering Equation (RE) [Kaj86] where both emitted radiance L_e and reflected radiance L_r contribute to outgoing radiance L_o. Many different formulations exist, e.g. integration over hemisphere or over surfaces, depending on the context of application. For a surface point x in direction ω, the RE becomes [Rit+12]

$$\begin{aligned} L_o(x, \omega) &= L_e(x, \omega) + L_r(x, \omega), \\ L_r(x, \omega) &= \int_\Omega L_i(x, \omega_i) f_r(x, \omega, \omega_i) \langle N(x) | \omega_i \rangle^+ \, \mathrm{d}\omega_i, \end{aligned}$$

where L_i is incoming radiance, $N(x)$ is the normal of x. f_r is for example a Bi-Directional Reflectance Distribution Function (BRDF) which describes how light is reflected from a surface point x given an incoming ω_i and outgoing ω direction. Note that this formulation is recursive since L_i depends on L_o, i.e.

$$L_i(x, \omega) = L_o(r(x, \omega), -\omega).$$

The ray casting function $r(x, \omega)$ returns the first surface hit point from x in direction ω.

However, the RE does not account for participating media. For volume rendering, the required Radiative Transfer Equation (RTE) [Cha60]

$$\begin{aligned} (\omega \cdot \nabla_x) L(x, \omega) = &- \overbrace{\sigma_t(x, \omega) \; L(x, \omega)}^{\text{absorption + out-scattering}} \\ &+ \underbrace{L_e(x, \omega)}_{\text{volume emission}} + \underbrace{\int_\Omega \sigma_s(x, \omega_i) f_p(x, \omega, \omega_i) L(x, \omega_i) \; \mathrm{d}\omega_i}_{\text{in-scattering}} \end{aligned} \tag{2.1}$$

includes absorption, out-scattering, in-scattering and volume emission. The extinction coefficient is $\sigma_t(x, \omega) = \sigma_a(x, \omega) + \sigma_s(x, \omega)$. In addition, the BRDF is replaced by a phase function f_p, e.g. for isotropic, Rayleigh or Mie scattering. The presented approach assumes isotropic scattering which makes these coefficients independent from position and orientation.

2.2 Ray Tracing

One way to solve the rendering equation is by using ray tracing. For each camera pixel, radiance is computed by shooting many rays in different directions. At each hitpoint, irradiance is recursively computed by shooting rays in many different directions until a recursion termination criteria is met and then averaging their results. With increasing number of rays and recursion depth, averaging converges to the exact solution. In the

context of GI, ray tracing natively supports a wide range of phenomena, including reflection, refraction and caustics. However, indirect illumination generates many incoherent rays, further degrading performance. Besides slow convergence when using Monte-Carlo based methods, particularly stochastic ray tracing introduces noise. Since rays are shot from camera pixels, stereoscopic rendering doubles the workload which can be ameliorated with scene based caching of intermediate computed irradiance.

[Löf+11] apply raytracing to a VR application. They combine raytracing with rasterization to interactively switch between a preview and a high resolution rendering with optional indirect illumination. Here the rasterizer is used for scene manipulation and application handling and uses shading to improve rendered images. Their raytracing implementation uses RTfact which achieves interactive refresh rates. However they report a rendering time of about 10 seconds when additionally computing indirect illumination on an old GPU.

Although current generation GPUs probably require less than a second up to interactive refresh rates, given the increase of computational power, this may not yet be practical for CAVE applications.

2.3 Photon Mapping

Classic Whitted style ray tracing is computationally very expensive to produce photorealistic images. But combining ray tracing with additional techniques produces more accurate results in less time. One such technique is photon mapping [Jen96] which computes GI in two passes. First, light sources emit photons which are bounced within the scene multiple times to yield a photon distribution, also known as photon map. Second, this distribution is used to compute irradiance for each camera pixel. For a given sample, this can be done by estimating density of nearby photons, or by gathering all visible photons. This technique simulates a wide range of phenomena and excels in rendering caustics [Rit+12]. [Pur+03; MM02] propose spatial hashing of photons to speed up density estimation. As soon as light changes, the photon map has to be recomputed. Depending on scene complexity, this can be costly. [Zho+08] use a k-d tree of scene elements for efficient ray shooting to improve performance of intersection tests. Recently [Yao+10] rasterize the scene from multiple perspectives to distribute photons. However choosing camera positions is difficult. In image space photon mapping [ML09], photons are emitted from a single point spotlight by using a reflective shadow map. In addition, they use a geometry shader to estimate the photon density.

[Hoa+10] combine image space photon mapping with irradiance caching and adaptive photon mapping for a CAVE application. In the first pass, they distribute initial photons as in [ML09]. In addition, they rasterize photon ellipsoids. To account for reflective and refractive surfaces, rays are shot towards these surfaces using OptiX™ framework[1]. In the second pass, photon gathering is approximated by shooting rays from surface hitpoints along their surface normals. Subsequently all intersected photon ellipsoids are gathered to the hitpoint. By using efficient rasterization and GPU shaders, their implementation achieves interactive framerates. For moderate scenes with 350k triangles, they report an actual framerate of about 4 Hz on a Quadro FX5800 when using downsampling to 800x800. For much smaller scenes, 10 Hz is reported. These performance numbers account for display synchronization and updates of distributed data as well, besides the computation of illumination.

[1] NVIDIA® OptiX™ is a programmable ray tracing engine (`https://developer.nvidia.com/optix`)

However, no user study is given about how 4 Hz is experienced by the user standing in the CAVE. Moveover many incoherent rays are shot to reflective and refractive surfaces which is inefficient compared to primary visibility rays from cameras or lights.

2.4 Precomputed Radiance Transfer

Precomputed Radiance Transfer (PRT) methods increase rendering performance by precomputing both the transfer function including visibility and the lighting function. For selected scene elements, these functions are computed, thus implying static geometry. The way these functions are represented significantly influences final rendering performance. [SKS02] proposed PRT and uses Spherical Harmonics (SH) basis to represent these functions. As a result, scene lighting can be efficiently computed using a dot product of a transport coefficient vector and a lighting coefficient vector since evaluation of dot products efficiently scales on GPUs. Even 4 coefficients of the first 2 SH bands already yield good results for diffuse illumination. However, high frequent illumination e.g. specular surfaces, requires more coefficients, thus significantly degrading performance. [NRH03] use non-linear wavelet basis functions and achieve interactive rates, but a non-linear basis requires a difficult GPU implementation [Rit+12]. In addition, Guassians [Gre+06] and radial basis functions [TS06] have been shown to work with PRT. Finally, [Rit+12] provide a detailed overview of PRT extensions and methods.

Depending on the amount of precomputation and accuracy of function encoding, a wide range of illumination effects are natively supported, including caustics and subsurface scattering. Although PRT requires a static scene in its initial form, extensions exist to support animated characters undergoing rigid transformation [Pan+07]. However, PRT is clearly not suitable for approximating illumination in fully dynamic scenes.

[Dmi+04] combine PRT with final gathering in the context of a CAVE application for car interiors. Since PRT techniques are best suited for low frequent surfaces, they stochastically integrate results from PRT using a Monte-Carlo technique for some scene sample points. Since stochastic integration introduces variance, sample directions need to be carefully chosen to reduce noise and flickering. They do this by using an importance criterion given by the BRDF of some scene surfaces. High frequent surface illumination is computed with expensive final gathering at a very low resolution of 400x300 pixels. To further improve rendering quality, they employ high dynamic range (HDR) environment maps which require an efficient tone mapping algorithm specificially tailored to the needs of a CAVE. To speed up tone mapping, they use a constant reproduction curve for all pixels. Furthermore, luminance adaptation conditions are computed for each screen depending on head tracking data. Despite the integration of tone mapping and final gathering, they report interactive framerates of 10 Hz with a Quadro FX3000G and a Dual XEON 3.06 GHz. However, dynamic scenes are not supported since geometry and PRT preprocessing requires about 100 minutes. Thus this approach is not suited for fast visualization of scenes in a CAVE.

2.5 Instant Radiosity

Similar to photon mapping [Rit+12], Instant Radiosity (IR) algorithms emit photons from light sources and bounce them inside the scene. This yields a coarse photon distribution. Here, photons are known as Virtual Point Lights (VPLs). In a second pass, VPLs are

gathered or scattered to surfels, thus approximating GI. As long as the number of VPLs remains small, usually a few thousands, efficient shadow mapping can be used resolving indirect visibility. For both VPL distribution and GI evaluation, the efficient rasterizer can be employed. Moreover glossy and multi-bounces can be supported, although one-bounce VPLs already render plausible low-frequent GI. However by using too few VPLs, artefacts may become present in complex scenes with many depth discontinuities, e.g. aliasing, too bright and too dark spots. Since most computation is done for rendering view-dependent GI gathering or scattering, IR algorithms are not optimal for stereoscopic rendering.

[Kel97] introduced IR. Here VPLs are emitted into the scene by using ray tracing combined with a quasi-random walk[Kel96]: a subset of all initial, not yet emitted VPLs is shot along a primary light ray into the scene. At the first hitpoint, the VPLs' radiance is attenuated and a subset recursively continues their path until no VPLs remain at a final hitpoint. To gather GI, the scene is rendered with shadows for each VPL being the only point light source. The resulting images are accumulated with equal weight. Due to averaged radiances, only low-frequent illumination can be rendered. Unlike PRT no precomputation of GI is required, thus this method supports dynamic scenes, although it severly depends on scene complexity.

[DS05] efficiently generate one-bounce VPLs by using the rasterizer. Here the scene is rendered as seen from a primary directional light or spotlight into 4 textures storing position, normal, depth and reflected radiant flux (figure 2.1). These textures comprise a Reflective Shadow Map (RSM) where each texel represents a one-bounce VPL. Note that an RSM is similar to a Geometry Buffer (G-Buffer), except that the latter only stores albedo for the RGB spectrum instead of reflected flux. The application presented in this thesis uses RSMs for light sources. To speed up GI computation, the scene is rendered once from camera perspective into a G-Buffer. For each camera pixel, GI is gathered from all VPLs while taking inverse distance and orientations of both VPL and camera surfel[2] into account. By using inverse distances, singularities near close surfaces result in visible bright spots [Kel97]. [Lai+07] clamp each VPL contribution suppressing short distance light transport to work around this problem. This however results in a biased solution suffering from darkened regions in cavities and corners [Dac+14]. [NED11] compensate bias in screen-space by first finding surface elements which potentially contribute to compensation. As close elements in world space are close in screen space as well, this step is done in screen space improving efficiency. Next the residual operator, i.e. the missing energy due to clamping, is applied recursively to both direct and clampled indirect illumination. Using 1 to 3 iterations already yields nearly indistinguishable results from unbiased solutions. But 3 steps at the highest quality take about 550ms with a Radeon HD 5870 and a resolution of 1024x768. Therefore this method is not applicable to a CAVE.

For complex scenes, e.g. a room with multiple separating walls, using only one-bounced VPLs results in too dark or even black areas, e.g. between two walls and the light source being on the other side of one wall. Bidirectional IR [Seg+06] solves this problem by emitting VPLs from both light source and camera. As these VPLs are bounced multiple times, GI solution for complex scenes is greatly improved. Moreover, this algorithm places more VPLs in areas contributing more to the scene as seen from the camera [Rit+12].

Visual quality can be further increased by incorporating indirect visibility. [Rit+08]

[2] In the remainder of this thesis, a camera surfel is considered a surface element of a single scene surface as seen through a camera pixel during rasterization. A light surfel corresponds analogously to a light pixel. In general a surfel is a surface element, i.e. a distinct subset of a single surface.

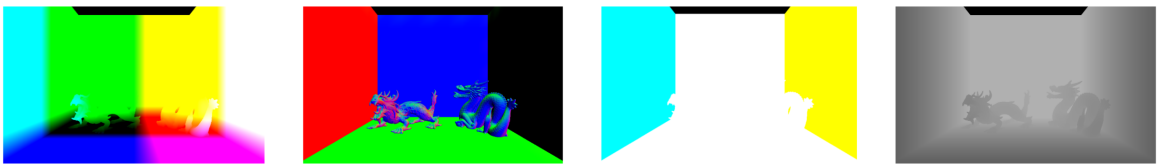


Figure 2.1: Layers of a RSM consist of position, normal, depth and reflected radiant flux textures (from left to right). Instead of the latter, the presented approach stores radiance in the third color buffer.

introduce Imperfect Shadow Map (ISM) to efficiently approximate visibility between two points in a scene. Initially, the scene surface is coarsely approximated by evenly distributed surface points. Given a number of VPLs created by some other method, these surface points are equally partitioned so that each partition maps bijectively to one VPL. When rendering surface points into their VPLs' Shadow Map (SM), no visibility is computed during rasterization. As these two approximations result in severe artefacts when rendering shadows, an additional pull-push pass significantly reduces SM holes visible as erroneous white pixels in the SM. During the pull phase, the image is downsampled by a factor of two multiple times. The resulting image is upsampled again during the push phase, which finally removes SM holes at the cost of reduced accuracy of the visibility term in the rendering equation. Further improving performance, the authors use one large ISM storing the SM for each VPL, thus limiting the number of VPLs. Concerning performance improvements, the authors compare the creation time of an ISM including pull-push phase (17ms) with the creation time of a SM (584ms), resulting in a speedup of about 34x using 256 VPLs and an ISM resolution of 256^2. For the Sponza scene, ISM creation takes about 52ms on an old NVIDIA GTX8800. Since ISMs are view-independent, they fit quite well to stereoscopic rendering. However, complex scenes with many details may require at least 1024 VPLs and an ISM resolution of 4098^2 for pleasing results. In addition, too dark or bright areas due to smearing of visibility holes still exist. Recently, [BBH13] improve ISM by using a tesselation shader of modern GPUs to create initial point sets in parallel. As these approximations become particularly noticable for glossy surfaces, [Hol+11] propose a method to avoid visibility holes when computing the VPLs' SM in one pass, optionally incrementally. In principle, resolving visibility between VPLs and surfels is sped up by arranging these VPLs in a Bounding Volume Hierarchy (BVH). A cut through this hierarchy separates visible from non-visible VPLs. Besides unimproved rendering for diffuse surfaces, it remains unclear to what extend this algorithm suits to VR environments, i.e. future work should be done to investigate the amount of temporal flickering realized by human observers. It requires a complex implementation as well. Another approach to compute visibility is presented by [Don+09]. Here VPLs are clustered into Virtual Area Lights (VALs) enabling use of efficient soft shadow mapping techniques to compute visibility. As point-to-point visibility queries are avoided, their algorithm does not suffer from banding artefacts.

When combining IR with indirect visibility by employing shadow mapping, the limited number of VPLs reduces the visual quality of rendering large complex scenes. To circumvent this limitation, Importance Sampling tries to position the VPLs more carefully in areas contributing more illumination to the final rendered image. [GS10] uses a propability based on the average image luminance to decide where to place VPLs. Their algorithm however introduces temporal flickering for dynamic light sources. As an alternative, flux distribution from the RSM to adapt the sampling pattern has been investigated by [DS06]. [BBH13] reduce

flickering by estimating importance of scene surface points visible from the camera. Although they ensure temporally coherent sampling, flickering is not avoided, reducing immersion and further increasing the chance of cybersickness due to abrupt change of perceived brightness levels. Therefore, Importance Sampling suffering from these issues should not be included in VR applications.

Instead of gathering all VPLs, [DS06] scatter them into a camera accumulation buffer using efficient rasterizer. For each VPL, a quadrilateral centered at the VPL's position in screen space is rendered to restrict the VPL's influence. This quadrilateral is known as a splat. Restricting the VPL's influence is justified by the observation that indirect lights significantly contribute to their direct neighborhood only. To account for this, the splat size is proportional to VPL intensity and inverse VPL-camera distance. Therefore the splat size can become as large as the screen size, potentially overdrawing the accumulation buffer with 1024 VPLs. Finally, GI for a screen pixel is computed depending on surface normal and VPL-surface vector. The presented algorithm restricts the splat size, thus introducing bias and leading to darkened areas [NW09]. [NW09] improve efficiency of splatting by preventing buffer overdraw and by employing hierarchies. Here a hierarchy of non-uniform, disjoint splats is initially build depending on normal and depth discontinuities of the current camera view. Therefore more computation is done is areas of more geometric details, leading to improved results. Next each VPL is splatted into all computed splats while using angles and distances. For scenes with complex discontinuities, this splatting scheme degrades performance as most splats are at their finest level. In this worst case, their algorithm degrades to non-hierarchical splatting. Furthermore, this approach requires a carefull upsampling strategy to avoid artefacts, e.g. blocks, haloing and ringing. Although being view-dependent, [Hoa+10] recently applied multi-resolution splatting to a CAVE environment. By using a cluster of 12 nodes, each equipped with a XEON 3.2GHz and a NVIDIA Quadro FX5800, a dynamic scene with 780k triangles renders at about 11 Hz while downsampling to 800x800.

2.6 Discrete Ordinate Methods

The evolution of light transport including participating media, absorption, scattering and emission is modelled by the Radiative Transfer Equation (RTE) (eq. 2.1). As this equation can be discretized in both space and orientation, discrete ordinate methods (DOM) transport light as energy exchange between neighboring lattice cells. The final light distribution within this lattice is then approximated by successive local propagations for all lattice cells. Such fine grained local operations fits to modern GPU architectures very well. In addition, DOM provides a good trade-off between accuracy and computational costs, and it does not restrict the nature of the underlying medium [Fat09]. However, as these local operations are repeatedly interpolated between neighboring cells, sharp light beams are smoothed which is known as light smearing [Fat09]. Thus DOM fits best to low frequent illumination. Moreover, angular discretization introduces the ray effect [CLP93] where propagation can cause distracting beams of light.

[Gei+04] use the Lattice-Boltzmann method to approximate the diffusion model of photons. During one propagation step, photon densities in each lattice cell are locally redistributed to new directions. Subsequently these redirected densities flow to neighboring cells. Here, propagation repeats until the system converges to a steady state. Besides stability and accuracy, the Lattice-Boltzmann method allows straightforward parallelization. Furthermore,

their implementation allows multiple anisotropic scattering as in clouds, dust, and smoke. But interaction between light and surfaces is not taken into account. Although this method delivers good visual results, it requires 1216Mb memory storage for a 256^3 lattice configuration, where each cells stores 19 directional densities as floats. At the time of writing, they reported a performance of 6.15 seconds per iteration step on an old Pentium 1.6GHz. Performance numbers for a GPU implementation are not available.

[Eva98] combines SH with DOMs (SHDOM). Here, SH is used to efficiently compute the scattering integral in the RTE. As long as scattering depends only on scattering angle, this yields a simple multiplication in SH space. Using DOM, these SH representations are stored in discrete lattice cells, further reducing memory and computational requirements. In order to increase accuracy, an adaptive grid has been implemented. Compared to spectral representation, using SH improved efficiency of simulating atmospheric radiative transfer, at the cost of introduced approximation errors.

2.7 Light Propagation Volumes

Another relatively new class of real time GI algorithms builds upon SHDOM: LPVs. They use techniques from both SHDOM and IR to combine their strengths and weaknesses. Light is efficiently propagated using SHDOM with usually two bands, while VPLs are injected from rendered primary light RSMs. As a result, these algorithms support fully dynamic scenes and approximate diffusion for low frequent light. However, with two SH bands they still suffer from light smearing and ray effect and can only support low frequent illumination. Since light distribution is stored in scene space, LPV fits well to stereoscopic rendering as in CAVEs.

[KD10] introduced the first method of LPV. During the initial light injection stage, they render a RSM for every light source to sample VPLs. These VPLs are transformed into SH representations to be stored in LPV cells. In order to avoid self lighting and shadowing, VPLs are moved by half the cell spacing in direction of their normals. This step is referred to as VPL injection. After distributing initial intensity in the scene, a geometry lattice needs to be initialized to approximate scene geometry. This grid is required to approximately compute diffuse shadows. Instead of iterating through meshes, they sample the scene's surfaces by using depth buffers, normal buffers and RSMs. Here, an optional depth peeling pass can be used to extract more geometry information from RSMs. For every extracted surfel, its blocking potential is estimated by using the surfel area. This propability and the surfel orientation are both expressed with blocker lattice functions $\mathcal{B}_{cc'_j}$ in SH space. These functions, encoded as 4 dimensional vectors, are accumulated to the VPL's destination grid cell. Next, light is propagated in the scene by using successive local iteration steps which is inherint to grid-based methods and DOMs. Here, light intensity L_c from a given lattice cell c is propagated to its 6 neighboring cells along its axial directions. For each face c_j of these cells, incident flux $\langle L_c|\Gamma_j\rangle$ is computed and transformed into outgoing intensity $\langle L_c|\Gamma_j\rangle\Gamma_j$ of the neighboring cell for subsequent propagation. To account for diffuse shadows, propagated intensity through faces needs to be attenuated by using occlusion propabilities

$$b_{cc'_j} = \langle \mathcal{B}_{cc'_j}|\Gamma_j\rangle$$

which are stored in the geometry lattice. This propagation scheme is used for all light intensity cells and iterations:

$$\Delta L_{c'_j} = (1 - b_{cc'_j})\langle L_c|\Gamma_j\rangle\Gamma_j$$

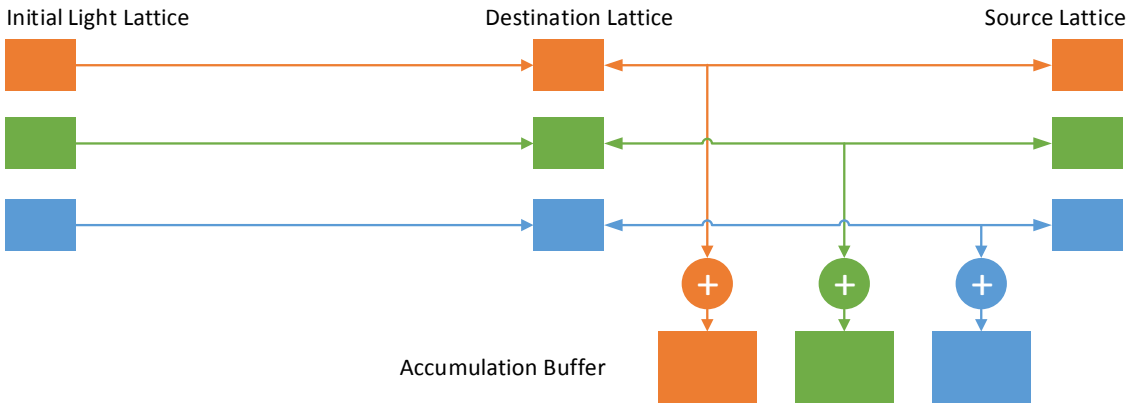


Figure 2.2: Light is propagated from source to destination lattice in round robinstyle (blue arrows). All intermediate solutions in destination lattice (pointed to by arrows) are accumulated in a third buffer (right) which holds the final result after n propagation steps. Initial propagation is often specially treated, e.g. by explicitly ignoring buzzy blocking. Each lattice holds a single color component in the RGB spectrum.

As this equation contains both L_c and $L_{c'_j}$, at least two lattices are needed for one propagation step: a source and a destination lattice (figure 2.2). A third accumulation lattice stores additively all intermediate distribution solutions from the destination lattice. Note that the first solution can be discarded. After all propagation steps, the accumulation lattice holds the final result. This approach however should not be used as soon as Γ_j are defined to be energy conservative because otherwise energy is created with each propagation step. Concerning the number of iterations, the authors propose a heuristic: two times the longest side of LPV similar to [Gei+04]. This heuristic ensures that initial light distribution reaches every lattice cell after propagation. For a 32^3 lattice, this evaluates to 64 steps. As each step takes about 1 ms, this heuristic is not practical for real time applications. Therefore they proposed Cascaded LPVs using multiple lattices with different cell size which is similar to a Level-of-Detail (LOD) approach. Although light can now travel larger distances, their approach is view-dependent. In classic LPV, 8 propagation steps already yield good results. Finally, light is rendered into image space using a camera G-Buffer being previously used for geometry injection. For each camera surfel, light intensity is queried from the LPV cells by trilinear interpolation of SH coefficients. In case scene geometry is static, the geometry lattice can be initialized once and then kept in constant memory during rendering. This is the case for the approach in this thesis. Since light intensities are stored outside of scene geometry, participating media is a natural extension to this algorithm [KD10]. In contrast to IR methods, significantly more VPLs can be sampled because they are only needed to initialize the LPV. This property is employed in this thesis experimenting with surface lights. Their algorithm also supports multiple light sources, area lights, environment maps and larger groups of point lights (e.g. particles). In case of environment maps, the outer lattice cells are initialized with light intensity from this map. According to [Loo+12], it produces smooth indirect shadows. However, when surfaces are smaller than the cell size, they might not produce shadows, e.g. foliage. But shrinking the cell size requires more iteration steps during propagation, which decreases performance significantly. The same holds true for large spatial distances [Loo+12]. Especially when no multi-resolution scheme is used, i.e. lattices of multiple resolutions near the camera, real time requirements may not be feasible

for reasonably sized LPVs and for medium large scenes since the number of iterations should be roughly two times the longest side of the LPV. In this case, the cell size should not be too small. If the grid resolution is chosen to be too coarse, self-illumination and self-bleeding artefacts can result. These recognized artefacts can be mitigated by introducing a dampening factor based on the direction derivative of intensity distribution. This however introduces bias seen as too dark areas. Thus this algorithm is best suited for low frequency diffuse lighting of small to moderate dynamic scenes with optionally multiple lights.

[Bør+11] modified the injection, propagation and rendering scheme to account for subsurface scattering. Instead of storing reflected flux in the RSM [KD10], they store the incoming flux to estimate the transmitted light intensity of VPLs inside an object. Snell's law and Frenel transmission factors are included in an extended directional intensity distribution function which is projected into SH to get their custom expansion coefficients. During propagation, they adapted the transfer functions to include extinction, i.e. the fraction of radiance being absorbed and scattered per unit length. For both absorption and scattering, they require additional lattices as they support heterogeneous media with varying extinction over space. In addition, each RGB channel may have a different extinction coefficient. After propagation, light distribution inside scene objects is converted into radiance estimates on the corresponding surfaces. Instead of directly using grid cells as point light sources within scene objects, they are approximated by uniform area lights. Each area light is parallel to the surface it points to, and it spans a cross section of the area of its parent grid cell. The authors found area lights superior because slight surface translation might lead to dramatic change in incident lighting, the grid resolution can change and a camera ray being refracted through surfaces will almost never hit light, according to the authors. Finally, this method should work well with highly scattering media since four SH coefficients approximate a strong diffusion process. However, the intensity of subsurface scattering greatly depends on the chosen grid resolution. As in [KD10], small scale features tend to be too dark as well. Moreover this method is not suitable for light transport in highly translucent materials, and it is not suited to preserve distinct features of light.

In contrast to [Bør+11], [BSA12] support multiple scattering in isotropic and homogeneous media using a modified propagation scheme. In addition, the light lattice is ray marched to render volumentric effects from multiple scattering, e.g. fog. In the injection stage, radiance from RSM is attenuated to account for participating media according to Beer's law:

$$I(x) = I_0 \cdot e^{-\sigma_t x} \tag{2.2}$$

Here, additional VPLs are injected between light source and scene geometry as seen from the light source. These VPLs emit single scattered radiance to the camera and are only used in the first propagation stage. Instead of using 6 neighbor cells and thus 30 neighbor faces [KD10; Bør+11], they directly propagate into 26 neighbor cells. The modified propagation reduces transfered radiance $\Delta L_{c'_j}$ by an extinction coefficent $\sigma_t = \sigma_a + \sigma_s$ being weighted by the average neighbor cell distance d, whereas extinction includes out-scattering and absorption. Furthermore, in-scattering $\Lambda^+_{cc'_j}$ is defined as a fraction of total out-scattered energy, where

this fraction depends on transfer vectors:

$$\begin{aligned} \lambda_s &= d \cdot \sigma_s, \\ \gamma_j &= \frac{\langle \Gamma_j | \mathbf{1} \rangle}{\sum_k \langle \Gamma_k | \mathbf{1} \rangle} = \frac{\Gamma_{j,x} \cdot 2\sqrt{\pi}}{\sum_k \Gamma_{k,x} \cdot 2\sqrt{\pi}} = \frac{\Gamma_{j,x}}{\sum_k \Gamma_{k,x}}, \\ \Lambda^+_{cc'_j} &= \gamma_j \lambda_s \langle L_c | \mathbf{1} \rangle \end{aligned} \tag{2.3}$$

This approximation holds true since in isotropic media, in-scattering is proportional to out-scattering. Finally, this in-scattering is added to the transfered radiance including normalized extinction $\lambda_t = d \cdot \sigma_t$:

$$\Delta L_{c'_j} = (1 - b_{cc'_j}) \left((1 - \lambda_t) \langle L_c | \Gamma_j \rangle + \Lambda^+_{cc'_j} \right) \Gamma_j \tag{2.4}$$

This equation follows the RTE (eq. 2.1) except that it is not recursive due to the assumption of one isotropic medium and that in-scattering does not depend on absorption. They additionally scale Γ_j because

$$\langle \Gamma_i | \Gamma_j \rangle \neq 0. \tag{2.5}$$

During rendering, they ray march the scene to evaluate illumination by participating media, i.e. one ray is shot into the scene per camera pixel up to the first hit point. Each ray samples light lattice cells, and the result is attenuated according to Beer's law. In terms of performance, this method has been shown to require 6ms for injection (256x256 VPLs), 2.5ms for 8 propagation steps and 14ms for ray marching, whereas all timings use a 32^3 lattice configuration, OpenGL, CUDA and GTX580. Without ray shooting, this method is well suitable for CAVE applications. Although only homogeneous media can be rendered.

Uniform lattice configurations have the advantage of fast and cache friendly access. Particular in sparse scenes, valuable memory can be saved by using non-uniform hierarchical lattices. [OD13] introduce an octree data structure for LPV, combined with an additional index structure. Here, propagation is similar to cascaded LPV [KD10], but it is done separately on each octree level. As a result, light is propagated farther on coarse levels, as in cascaded LPV. However, this octree light data structure cannot be used directly during rendering as this would require access to all octree and index levels which does not map well to GPUs. Instead, this octree is merged into an uniform lattice enabling efficient rendering. Moreover, due to hierarchical data structure, this method requires less iterations steps of 4 instead of 16 compared to cascaded LPV for similar results. Finally, the authors found this method slightly faster concerning performance. However, an octree introduces an additional overhead for creation, update and traversal which might not fit best to GPUs. Besides increased complex implementation, they detected small intensity variations between octree levels. Although they found them to be not disturbing, these variations might actually introduce flickering. Consequently, this method might not fit well to CAVE applications and further investigations are required.

3 Concept and Design

This chapter presents the rendering pipeline concept from a general perspective having an apparent emphasis on mathematical formula and theoretical concepts without implementation details. It brings together distinct algorithms to finally constitute the renderer.

After introducing the concept to render in a multi-display environment, this chapter explains initial procedures to properly precompute the geometry lattice as well as complex constants during rendering. Transfer vectors to distribute light between lattice cells need to be generated and scaled to approximately adhere to the physical law of energy conservation. These are approximated due to limited floating point precision of data during light propagation. Outscattering fractions (eq. 2.3) are precomputed as well to speed up computation during rendering.

Afterwards focus is given on the runtime behavior, e.g. to inject primary light into the scene lattices, to propagate illumination between lattice cells, to compute both indirect and direct illumination and finally to correctly send scene radiance to the display adapter by employing tone mapping and gamma correction. The latter two are required to convert high precision radiance values to proper display pixel values based on the display color space. This chapter presents experiments with surface lights as well, although with apparent limitations. Finally, the novel incremental approach to reuse light distributions from previous frames is explained being accompanied with an evaluation of differences to previous approaches in this field.

3.1 Multi-Display Environment

The renderer is designed to work in a multi-display installation, e.g. a powerwall and a CAVE. This cluster consists of a single master node controlling multiple render nodes, at which each node may serve one or more display devices. Each render node receives projection parameters computed by this framework being based on the cluster configuration, on the render node and whether to render left or right eye in case of stereoscopic rendering. Here off-axis projection is employed for rasterizing the virtual scene.

Communication between master node and render nodes is limited to a client-server approach. The master node additionally handles user input, e.g. but not limited to wand, head tracking, keyboard and mouse. While the application is almost entirely replicated on all render nodes[1], parameter updates are incrementally distributed in a synchronized manner to ensure consistent rendered results between displays. This incremental approach saves bandwidth and it helps to minimize synchronization time.

[1]The master node additionally features behavior concerning bookkeeping, input handling, frame starting and finishing and sending incremental parameter updates.

3.2 Initialization

Before rendering can take place, the renderer needs to be initialized. This step involves creating both renderer and resource objects, resolving dependencies, and passing a loaded scene configuration file to each configurable instance for self initialization. The latter is done in-order to respect dependencies. For example, it includes filling the geometry lattice, transfer functions and other precomputable values during propagation.

3.2.1 Geometry Lattice

Since only static scene geometry is supported, the geometry lattice is filled during the initialization phase. Here geometry is injected by adaptively sampling surfaces based on triangle area and cell size (algorithm 1). This ensures that each triangle contributes to each intersected lattice cell, thus preventing holes in the geometry lattice.

Algorithm 1: Scene geometry injection

```
foreach scene triangle face f do
    apply model transform to vertices v_0, v_1, v_2;
    compute triangle area;
    uniformly sample points within quadrilateral;
    discard sample points being outside of triangle △v_1 v_2 v_0;
    foreach sample point s do
        compute geometry lattice cell;
        skip out-of-lattice samples;
        compute SH coefficients c_coeff from normal N(s);
        inject c_coeff;
```

Furthermore, the sample points are ensured to be uniformly distributed with respect to each triangle preventing bias and lattice holes. A uniform pseudo-random distribution of samples s within a triangle can be easily computed with barycentric coordinates and by

$$s = r_u \cdot v_1 + r_v \cdot v_2 + (1 - r_u - r_v) \cdot v_0,$$

where r_u and r_v are uniform pseudo-random variables in the interval $[0, 1]$ and where $r_u + r_v <= 1$ holds true to discard points being outside of the triangle.

Next, SH coefficients c_{sh} are computed from each sample's normal $N(s)$ in direction ω and additively injected into the geometry lattice. This corresponds to an update of the cell's blocking distribution function $\mathcal{B}_{cc'_j}$ which expresses the amount of light blocking in a given direction ω. The larger $\langle \mathcal{B}_{cc'_j} | \mathrm{SH}(N(s)) \rangle$ evaluates, the more light can travel in ω.

In contrast to [KD10], the geometry lattice is not offset by half the cell spacing since a different propagation scheme of 26 neighbor cells instead of 30 neighbor faces is employed. Here $\mathcal{B}_{cc'_j}$ of the source cell c is directly used during the propagation steps to account for indirect shadows.

An alternative solution might be to directly inject sample normals into the geometry lattice, and subsequently normalizing the lattice. Using this approach, the propagation step has to compute the clamped cosine angle between the mean lattice cell normal and the propagation direction. This however is a very coarse approximation, resulting in too bright indirect shadows.

3.2.2 Light Lattice and Propagation

Next the light lattice renderer is initialized managing light propagation in the scene. During initialization,

1. arrays, buffers and lattices for host, OpenGL and OpenCL or CUDA are allocated,
2. transfer vectors are precomputed and scaled,
3. outscattering fractions are precomputed.

In case of propagation without both blocking and absorption, energy should be conserved. This puts an additional constraint on Γ_j such that the total propagated energy to 26 neighbors equals the source cell's energy:

$$\sum_j \langle\langle L_c|\Gamma_j\rangle|\mathbf{1}\rangle \stackrel{!}{=} \langle L_c|\mathbf{1}\rangle.$$

Following this constraint and eq. 2.5, Γ can then be computed with sampling and subsequent scaling. Here, a transfer vector Γ_j from a source cell c to a given neighbor cell c_j is computed by sampling points in c_j. Samples outside the visibility cone from c to c_j are discarded, being known as stochastic rejection sampling. Then directions from source cell to sample s are normalized and used to compute mean cosine lobe SH coefficients Γ. This procedure is repeated for each of the 26 neighbor cells.

These transfer vectors additionally need to be scaled to avoid significant energy loss or creation after each propagation iteration. Here transfer vectors are scaled by setting up several random initial radiance distributions for one source cell. These are propagated to 26 neighbor cells by using a sampled scale factor κ, and energy differences ΔE are averaged. The final κ is found by solving a minimization problem concerning ΔE. In detail, this step is repeated for several scale factors within a given initial interval $\mathcal{I}$, e.g. $[0.1, 0.4]$, whereby the factors are found with golden section search [Kie53]. For $x_2 \in [x_0, x_1]$ and $x_0 > 0$, x_2 is found with

$$\begin{aligned} x_2 &= x_1 + \frac{x_1 - x_0}{1+\varphi}, \\ \varphi &= \frac{b}{a} = \frac{x_1 - x_2}{x_2 - x_0} = \frac{1+\sqrt{5}}{2}, \\ x_3 &= x_0 + b. \end{aligned}$$

The interval for the next iteration is

$$\mathcal{I}_{\text{next}} = \begin{cases} [x_0, x_2] & , |\Delta E(x_3)| > |\Delta E(x_2)| \\ [x_2, x_1] & , \text{otherwise} \end{cases}$$

and the scale factor is then

$$\kappa = \begin{cases} x_2 & , |\Delta E(x_3)| > |\Delta E(x_2)| \\ x_3 & , \text{otherwise.} \end{cases}$$

Finally the search terminates if

$$x_1 - x_0 < \tau \cdot (x_1 + x_0),$$

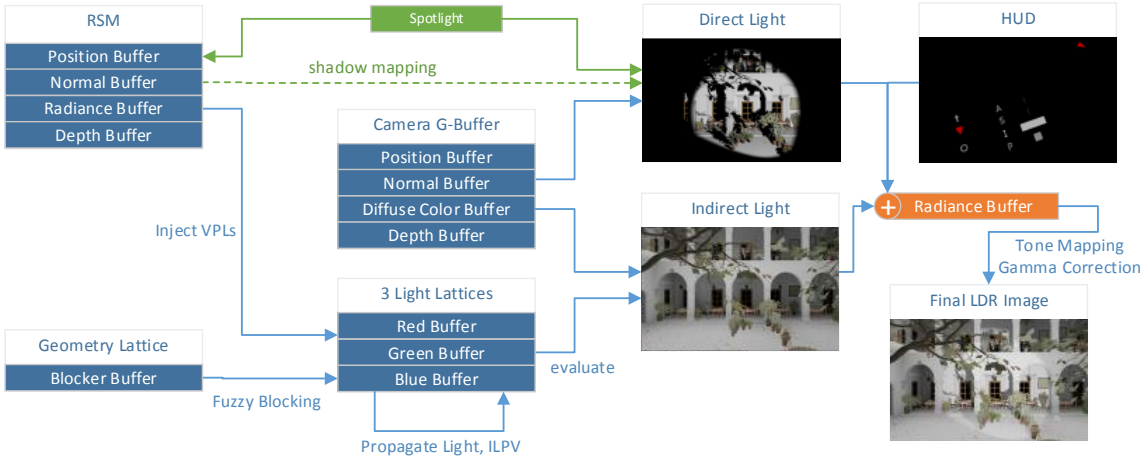


Figure 3.1: Principal rendering workflow from left to right. Initial rendered RSM and Geometry lattice are employed for VPL injection and light propagation with fuzzy blocking. The camera G-Buffer is used for deferred shading of direct and indirect illumination, resulting in HDR radiance values being stored in a single accumulating radiance buffer. Both tone mapping and gamma correction fit the HDR values into displayable LDR values to be sent to the display adapter.

where τ is the tolerance of about 10^{-6}. In contrast to linear search, golden section search is significantly faster. Depending on τ and I, only 10 - 20 different factors are probed to obtain an energy loss of about 10^{-4} to 10^{-6} during one propagation to 26 neighbors without blocking and absorption. Although these computations are not real time, they are only needed when the lattice configuration changes.

Next, outscattering fractions γ_j to neighbor cells are precomputed as in eq. 2.3. Note that the transfer vector's x-component $\Gamma_{j,x}$ corresponds to energy and that $2 \cdot \sqrt{\pi}$ to obtain energy [BSA12] cancels out.

3.3 Runtime

After all renderers and resources have been created and initialized, rendering the virtual scene in the CAVE installation can be conducted. As in [KD10], a RSM is rendered for each spotlight for VPLs to be injected into 3 light lattices (figure 3.1). The geometry lattice can be rendered once for static scenes, or updated every frame for dynamic geometry. Its stored blocking factors are loaded during propagation to account for approximate indirect shadows. An additional camera G-Buffer speeds up rendering in a deferred shading pass for both direct and indirect illumination where the latter employs propagated light distribution in scene space. This pass returns radiance values being out of displayable color range, better respecting dark regions. Thus both tone mapping and gamma correction map these radiance values to properly displayable values for the currently chosen display color space and environmental lighting. This step may only be applied as the final step on a single buffer, just before sending the result to the display adapter.

3.3.1 VPL Injection

As in [KD10], VPLs are created from a previously rendered RSM for each spotlight, and then additively injected into the light lattice for each RGB spectrum component. The VPLs are also moved by half the cell spacing d_s in direction of its normal n to prevent self illumination (algorithm 2).

Algorithm 2: VPL injection from RSM

foreach *texel* $t \in RSM$ **do**
- set normal $n = N(t)$;
- set position $p = P(t) + n \cdot (0.5 \cdot d_c)$;
- compute $c_{\text{coeffs}} = \text{SH}_{\cos}(n)$;
- scale $c_{\text{coeffs}} = c_{\text{coeffs}} \otimes C(t)$;
- compute lattice cell c from p;
- additively inject c_{coeffs} into c;

Here cosine lobe SH coefficients are used since they best approximate the visible hemisphere of a VPL when using only two SH bands [KED11]:

$$\text{SH}_{\cos}(\omega) = \begin{pmatrix} c_0 \\ -c_1\omega_y \\ c_1\omega_z \\ -c_1\omega_x \end{pmatrix}, \quad c_0 = \frac{\sqrt{\pi}}{2}, \quad c_1 = \sqrt{\frac{\pi}{3}}.$$

3.3.2 Propagation

In contrast to previous LPV approaches, only a source and a destination lattice is used instead of an additional accumulation buffer (compare figure 2.2). The reason is that transfer vectors are defined and scaled to be energy conservative. After one propagation step over all lattice cells, the total energy in all cells should be approximately constant without considering absorption and blocking. Therefore simply accumulating all intermediate results from previous propagation steps creates energy. Averaging the final result in the accumulation lattice with the number of propagation steps mitigates energy creation, but also counteracts effective light propagation. The reason is twofold: At the beginning, light distribution is spatially concentrated due to VPL injection and smears out during propagation. But a mean average neglects smaller values in favor of larger values, thus smaller radiance values after a large number of iterations are generally neglected, and radiance values from first iterations receive the most weight. Therefore, more propagation steps are required for good results. In addition, the proposed ILPV approach works best without an accumulation lattice.

As in [BSA12], light is propagated to 26 neighbor cells. This is done for each RGB spectrum component and each lattice cell constituting one propagation step from source to destination lattice. A single propagation to one neighbor cell follows the propagation scheme in eq. 2.4 except that extinction, absorption and scattering coefficients are differently computed.

Each scattering σ_s and absorption σ_a coefficient is the probability of being scattered and absorbed, respectively. Therefore the codomain is $[0, 1]$. But these coefficients have to be normalized since the lattice discretizes scene space. [BSA12] obtain a normalized scattering coefficient λ_s by multiplying with the mean neighbor cell distance in world space. This

however violates the property of being a probability. Instead, the normalization

$$\begin{aligned}\lambda_a =& {\sigma_a}^{\overline{d}_{cc_j}},\\ \lambda_s =& {\sigma_s}^{\overline{d}_{cc_j}},\\ \lambda_t =& \lambda_a + \lambda_s,\end{aligned}$$

ensures that both σ_a and σ_s remain probabilities. The mean neighbor cell distance can be approximated as

$$\begin{aligned}\overline{d}_{cc_j} =& \frac{d_s}{26}(6 + 12\sqrt{2} + 8\sqrt{3})\\ \approx& 1.4164218 \cdot d_s,\end{aligned}$$

where d_s is the cell spacing.

3.3.3 Indirect Illumination

After light has been propagated in the scene, indirect illumination can be rendered from any camera perspective, e.g. left and right eye. Thus LPV methods fit well to stereoscopic rendering.

Here indirect illumination is computed in a deferred shading pass by using a camera G-Buffer (algorithm 3). For each camera surfel, lattice position p in lattice space $[0, 1]$ is obtained. For each RGB spectrum component, p is used to tri-linearly interpolate SH coefficients from 8 nearest lattice cells, yielding a radiance distribution. Multiplying with cosine lobe SH coefficients corresponding to the surfel's normal results in exitant radiance approximating indirect light.

Next, the indirect illumination radiance is properly mixed with the surface color under white light. The problem is that to be physically more accurate, the RGB color space is a too coarse representation for color computations [PH10, p. 261-279]:

1. Displays have different Spectral Power Distributions (SPDs), i.e. the amount of light at each wavelength additionally depends on the physically properties of the display panel. LCD, LED and Plasma Displays may also perform internal non-linear color mapping.

2. The spectral matching curves XYZ, determined by CIE, are too coarse basis functions for spectral computations. Instead the product of (x, y, z) values should be done in more accurate spectrum representations, e.g. by discretizing spectrum into uniform intervals. By using XYZ and RGB spectral response curves, the (r, g, b) values can be computed depending on both SPD and display properties.

3. Computations should be also be based on standardized SPDs, e.g. D65 representing midday sunlight.

However, these more accurate color computations are still too expensive for real time CAVE applications. Therefore, the implementation simply bases computations on (r, g, b) values since they are computationally and storage efficient. Mixing indirect light with surface color is then computed by a piecewise vector multiplication.

Due to spatial discretization from the lattice, resulting radiance is divided by uniform cell area. The reason behind this approximation is that the RSM forms a 2-dimensional cut through the uniform lattice consisting of cubes.

Algorithm 3: Light Lattice Evaluation

```
foreach camera surfel t in G-Buffer do
    set normal n = N(t);
    set position p' = P(t) + n(0.5 · d_s);
    set surface color c = C(t);
    convert to lattice space p ← p';
    compute c_coeffs = SH_cos(−n);
    foreach color component λ do
        tri-linearly sample radiance distribution L^SH_{t,λ} = C_{lattice,λ}(p);
        compute radiance L_{t,λ} = ⟨c_coeffs | L^SH_{t,λ}⟩;
        divide radiance by cell area d_s^2;
    mix radiance with surface color: out = √(max L_t) ⊗ c;
```

3.3.4 Direct Illumination

Direct illumination approximates light paths of the form $L(D|S)E$ where primary light L hits a surface which diffusely D or specularly S reflects it to the eye E. Diffuse reflectance is often modelled by body reflectance[2], statistically simulating phenomena that occur under the object's surface. As an example, it approximates local subsurface scattering where light enters a medium through an optical discontinuity, scatters and is partially absorbed, and finally exits this media where the light entered. Compared to global subsurface scattering, the path length between entrance and exitance lies within the microscale, i.e. it is smaller than a pixel size. Such microscale phenonema are commonly modelled by BRDFs that determine how light is reflected on surfaces as seen from the macroscale. In real time rendering, Lambertian BRDF efficiently accounts for body reflectance[AHH08, p. 228]:

$$f(\mathbf{l}, \mathbf{v}) = \frac{\mathbf{c}_{\text{diff}}}{\pi}. \tag{3.1}$$

Microscale surface roughness in contrast is not rendered by body reflectance, but by surface reflectance. Most real surfaces are not optically flat since their microgeometry[3] consists of microscale irregularities. In microfacet theory, each such irregularity, or microfacet has its own microscopic surface normal $\mathbf{h}$, being statistically distributed around a single macroscopic surface normal seen by the observer. The microfacet is assumed to be a perfect flat Fresnel mirror. It reflects light $\mathbf{l}$ to the eye $\mathbf{v}$ along its half vector $\mathbf{h}$. This is the reason why BRDFs derived from microfacet theory use a half vector, not the macroscopic surface normal $\mathbf{n}$[4]. Surface smoothness can then be expressed as a probability distribution of $\mathbf{h}$ around $\mathbf{n}$. In real time rendering, efficient BRDFs often denote m for smoothness, thus omitting evaluation of complex statistical functions. In addition, a BRDF should be normalized for parameters to be equivalent to reflectance values, e.g. c_{spec}[AHH08]. The half-vector-based normalized Blinn-Phong BRDF[AHH08, p. 257] delivers a good tradeoff between accuracy and efficiency:

$$f_{\text{BRDF}}(\mathbf{l}, \mathbf{v}) = \frac{\mathbf{c}_{\text{diff}}}{\pi} + \frac{m+8}{8\pi}\mathbf{c}_{\text{spec}}\ \overline{\cos}^{m}\theta_h. \tag{3.2}$$

[2] Directional-hemispherical reflectance measures the amount of reflected light from a given incoming direction, i.e. $R(\mathbf{l}) = dB \cdot dE^{-1}(\mathbf{l})$. It is a probability function for energy conservation.

[3] In detail, microgeometry lies in the scale between light wavelengths and pixel size.

[4] See [AHH08, p. 250] for a comparison between half vector and reflection vector based BRDF.

Applying the precise definition of the BRDF to non-area light sources and solving for radiance, the above equation can then be inserted into the rendering equation for direct illumination:

$$L(\mathbf{v}) = f_{\text{BRDF}}(\mathbf{l}, \mathbf{v}) \otimes E_L \cdot \overline{\cos}\theta_i,$$

where θ_i is the angle between $\mathbf{n}$ and $\mathbf{v}$. In phyiscally based rendering, irradiance is computed from intensity[5], i.e. $E_L = I_L r^{-2} = I_L f_{\text{dist}}(r)$. In real time rendering, this distance function often differs to allow fine grained control for artistic needs[6], which complicates setup though. The distance attenuation as in eq. 2.2 has the good property that irradiance can not become infinitely large for near surfaces, although it never reaches zero. The latter increases realism which is worth the additional computational cost. The implementation uses this function, in combination with a more flexible spotlight attenuation from Microsoft Direct3D, approximating umbra and penumbra [AHH08, p. 221]:

$$f_{\text{spotlight}}(\mathbf{l}, \omega_l, \theta_{\text{fov}}, \theta_{pu}) = \begin{cases} 1 & \theta_s \geq \theta_p, \\ \left(\frac{\theta_s - \theta_u}{\theta_p - \theta_u}\right)^{s_{\text{exp}}} & \theta_u < \theta_s < \theta_s, \\ 0 & \theta_s \leq \theta_u, \end{cases}$$

$$\begin{aligned} \theta_s &= \langle \omega_l | - \mathbf{l} \rangle, \\ \theta_p &= \cos(0.5 \cdot \theta_{\text{fov}} - \theta_{pu}), \\ \theta_u &= \cos(\theta_{pu}), \end{aligned}$$

where $\mathbf{l}$ is the normalized direction from surfel to spotlight position, ω_l is the spotlight's view direction, the spotlight's field-of-view is θ_{fov} and θ_{pu} being the size of penumbra.

The implementation also employs Percentage-Closer Filtering (PCF) for simulating direct hard shadows. In a previous stage, a RSM has been rendered for each light, containing a depth buffer. By projecting the surfel position into the light's NDC space and then comparing with the stored depth component, the visibility function $V(p, \mathbf{l})$ is easily and efficiently solved. However, a depth bias of about 5^{-4} roughly compensates for the limited IEEE floating-point single precision, i.e. it tries to counteract z-fighting seen as self-shadowing of surfaces[Eis+11]. For low resolution buffers, aliasing of shadow silhouettes can become intense. PCF improves upon this by testing depths for each pixel in a small window before applying a filter kernel. It returns a value within $[0, 1]$, where 0 equals to a blocked shadow sample.

Finally, irradiance is computed as:

$$E_L = I_L \otimes f_{\text{spotlight}} \cdot f_{\text{beers-law}} \cdot V(p, \mathbf{l}).$$

3.3.5 Surface Lights

So far, VPLs have been utilized in approximating one- and multiple-bounce indirect illumination by distributing them in the scene[7]. Moreover, VPLs can be positioned and oriented in a way to roughly compute complex surface lights with arbitrary shapes. This is made possible because VPLs comprise a simple geometric shape.

In detail, a possible solution is to position one VPL for each mesh vertex, or for each triangle. This thesis investigates the latter positioning approach. Here one VPL is placed on

[5] Note that for point lights, $I_L = \Phi(4\pi)^{-1}$.

[6] See [AHH08, p. 218-221] for an overview of some approximations.

[7] VPLs have also been used to simulate dense volume scattering, e.g. fog being illuminated by a spotlight[BSA12], or even subsurface scattering [Bør+11].

each triangle centroid, oriented along the triangle normal. This is efficient and maps well to current shading capabilities. However care must be taken when injecting these VPLs since their resulting radiance values have to be normalized to respect both mesh resolution and triangle area. That is, the mesh is supposed to emit a defined flux being converted into intensity for each triangle, by dividing with triangle count and by multiplying with triangle area. Subsequent computation is analog to injecting from spotlights.

Note that this approach is a rough approximation, but already yields acceptable to good results. However, it only accounts for indirect illumination. Direct illumination for these lights should be performed in a separate pass. In practice, when rendering surface lights without direct illumination, their shapes appear too dark compared to expected renderings, although they are not entirely black, depending on lattice configuration and mesh size.

3.3.6 Final Pass

The final pass consists of both tone mapping and gamma correction to map out-of gamut values into the displayable range (figure 6).

Tone Mapping

Both direct and indirect illumination are additively stored in a final High Dynamic Range (HDR) image to hold camera surfel radiance values. These values potentially comprise a very large range. However, optic displays are still limited in the range and precision of displayable luminance compared to virtual sensors in the renderer [AHH08]. To fit the wide range of illumination levels into the display's limited gamut, the potentially large physical radiance values are normalized to within the $[0, 1]$ range. This range standardized by the sRGB and AdobeRGB color spaces. In CAVE installations, tone mapping should be view independent, temporally coherent and efficient. The mapping "maximum to white" is not coherent between CAVE projections since each of them views the scene from a different perspective. Dark features, e.g. indirect illumination, also become unnoticable. In non-coherent "histogram equalization", the maximum bar depends on the maximum radiance value. Similarly mappings using log-average radiance[8] are generally not coherent as well[9]. However the local tone operator [AHH08, p. 479]

$$\overline{L}(x, y) = \frac{L(x, y)}{1 + L(x, y)} \tag{3.3}$$

fulfills the above requirements. It is efficient to evaluate, scales well to large resolutions and most importantly it is independent of the actual radiance distribution. It also has the advantage that low radiance is almost untouched, while high radiance gradually approaches 1 after applying this map. Nonetheless this nonlinear tone operator accentuates ropeing artifacts along antialiased edges. The implementation uses eq. 3.3 without a maximum radiance to simplify scene setup. It also ensures that no out-of-gamut colors are produced.

[8] The log-average radiance from the current and previous frame can be used for approximating light adaptation of the human eye, e.g. when entering a particularly brighter or darker room. Coherent implementation for CAVE installations is not trivial though.

[9] This however depends on the tone mapping function, and whether the differences are noticable.

Gamma Correction

After tone mapping, the renderer converts normalized Low Dynamic Range (LDR) radiance into gamma corrected nonlinear frame buffer values to be sent to the optical display. The limitations of display device technology and the surround effect[10][Poy03] both make this additional step inevitable in case improved realism is requested. Without gamma correction, banding artifacts near low radiance are accentuated due to reduced floating point precision[11] for encoding color values. As an example, the Quadro 6000 and the successor K6000 both support up to 10 bits per color component[12], while the implementation in OpenGL encodes each component as an IEEE single precision floating point value[13].

Three nonlinear transfer functions exist to uniquely define the map between normalized scene radiance, encoded pixel values for transport to the display device and display radiance to be percieved by the observer [AHH08, p. 141-143]. While each function depends on a distinct gamma, only 2 gamma values are required because the encoding gamma γ_{enc} is computed from both display gamma and end-to-end gamma:

$$\gamma_{\text{enc}} = \frac{\gamma_{\text{e2e}}}{\gamma_{\text{dsp}}}.$$

Rendering systems typically store results in sRGB or in AdobeRGB color space, having a $\gamma_{\text{dsp}} \approx 2.2^{-1}$. Power functions approximate these transfer functions which comprise of a power function segment and a linear segment for small values [Poy03, p. 257-280]. The latter prevents noise from becoming too noticable, e.g. in Rec 709 for HDTV or in sRGB for personal computers. In case the rendered image is almost free of noise, a pure power function x^{γ} suffices, having the advantage of increased evaluation performance. Thus the implementation simply employs

$$f_{gc}\left(\overline{L}(x,y)\right) = \overline{L}(x,y)^{\gamma_{\text{enc}}}$$

to convert normalized scene radiance into encoded pixel values. Due to the surround effect, γ_{e2e} should be adapted ensuring predictable emitted display radiance seen by the observer. Table 3.1 lists a recommendation for γ_{e2e}. This gamma affects both percieved contrast and brightness. An inappropriate value of 1.125 in a dark environment would make the rendering appear having less contrast and being too bright because $\overline{L}(x,y)^{0.6} < \overline{L}(x,y)^{0.45}$ for a display gamma $\gamma_{\text{dsp}} = 2.5$.

3.4 Incremental LPV

As [KD10] pointed out that using two times the longest side of the light lattice for the number of propagation steps is unfeasable for real time frame rates (figure 4). As such,

[10] Optical displays are may be positioned in an environment filled with external illumination. In this case, actual emitted display radiance may be differently percieved by the observer due to the human sensitivity and adaptation to brightness, e.g. in theaters and in offices. In particular in CAVEs, projected radiance is physically propagated inside the CAVE, further influencing percieved illumination[Bim+06].

[11] The actual precision also depends on the display capabilities and on the display adapter, e.g. DVI, dual-link DVI, HDMI, DisplayPort. The GPU device and its driver is also important.

[12] See NVIDIA GPU specifications at `http://www.nvidia.com/object/quadro_fx_product_literature.html`

[13] See specification of GL_RGBA32F

Environment Lighting	γ_{e2e}
dark (cinema)	1.5
dim	1.25
light (office)	1.125

Table 3.1: End-to-end gamma for typical ambient lighting conditions [Poy03, p. 85]. This gamma roughly compensates the surround effect.

propagation iterations have to be limited to approximately 16 steps (32^3) or 8 steps (64^3). In particular for higher lattice resolutions, light is effectively propagated small scene distances only reducing image quality which becomes apparent after comparing figures 4 and 5. By using light distributions from previous frames, light can be farther distributed, thus finally propagating through the entire lattice with a small number of iterations per frame. This increases rendering quality, or it saves propagation steps for same visual quality.

The proposed incremental approach uniformly scales both light distribution from previous frame L^{i-1} and initial light distribution from current frame L^i for each cell c:

$$L^i_{c'} = \widetilde{\lambda}_a L^{i-1}_c + \widetilde{\lambda}_b L^i_c. \tag{3.4}$$

Both scale factors are derived from λ_{incr} determining the amount of incremental rendering. Furthermore, scaling should conserve energy, even if initial light intensities change over frames, i.e.:

$$E^i \overset{!}{=} \widetilde{\lambda}_a E^{i-1} + \widetilde{\lambda}_b E^i.$$

Using this constraint, they are computed as

$$\begin{aligned}\widetilde{\lambda}_a &= \frac{E^i}{E^{i-1}}\lambda_{\text{incr}},\\ \widetilde{\lambda}_b &= 1 - \lambda_{\text{incr}},\end{aligned}$$

since

$$1 = \underbrace{\widetilde{\lambda}_a k}_{\lambda_{\text{incr}}} + \widetilde{\lambda}_b, \quad k = \frac{E^{i-1}}{E^i}$$

where $\lambda_{\text{incr}} \in [0, 1]$ and $\lambda_{\text{incr}} = 0$ means no incremental propagation.

Note that for best results, transfer vectors need to be properly scaled to approximate energy conservation in the absence of absorption and blocking. This incremental scheme is also independent of the propagation scheme, e.g. eq. 2.4, i.e. it can be easily applied to existing implementations. In particular, absorption and blocking are not included here to achieve temporally coherent rendering. Moreover, energy is used from spotlight's properties, instead of reading the whole lattice to improve performance. Due to eq. 3.4, the first frame needs to be rendered with $\lambda_{\text{incr}} = 0$ to obtain defined results and to prevent temporal lag at the beginning.

Due to this incremental formulation, differences become particularly noticable for few propagation steps (figure 3). Here 4 steps provide good results. Using a large λ_{incr}, one iteration already renders similar images at the cost of substantial temporal lag. Moreover 8 iterations with $\lambda_{\text{incr}} = 0.8$ produce images comparable to non-incremental rendering with

32 iterations which is shown by the mean absolute errors (MAEs) (figure 2). Since light is propagated farther than in LPV, this algorithm has its strengths for distant and dark areas (figure 3, 7). Although distant light is darker than after propagating a high number of steps with LPV, this circumstance is mitigated after applying tone mapping to intermediate HDR results.

However, ILPV introduces a small temporal lag (figure 3.2) until light distribution equilibrium is approximately logarithmically reached (figure 3.3). This delay becomes noticable for $\lambda_{\text{incr}} > 0.9$ and a lattice resolution of at least 64^3 which is caused by the propagation implementation if frame rate is significantly dropped below approximately 30 Hz. For higher frame rates, it is not noticable.

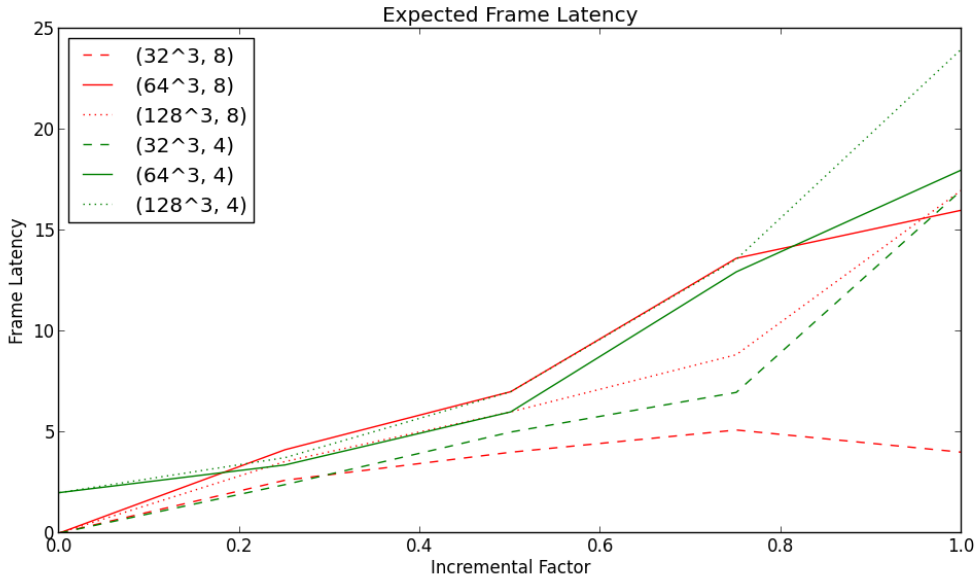


Figure 3.2: Frame latency plotted against λ_{incr}. Latency between 4 and 8 iterations does not significantly differ. 1 iteration introduces a noticable temporal lag.

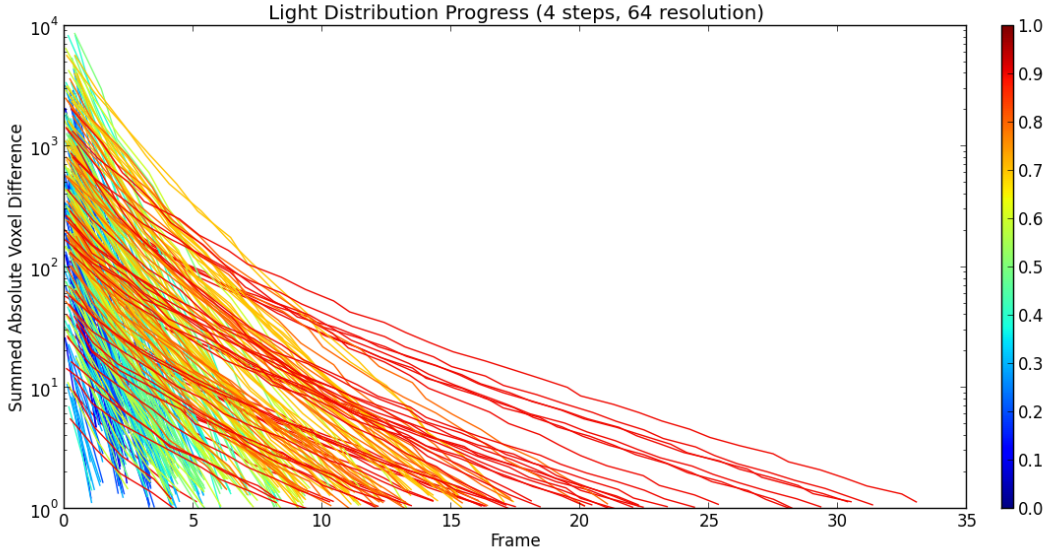


Figure 3.3: Logarithmic light distribution progress until equilibrium based on summed absolute voxel differences for each RGB component. Convergence depends on λ_{incr}, seen as slope of plotted graphs, as well as on the initial light change. During measurement the light abruptly moved 1.4m along near and far surfaces and it waited until equilibrium.

4 Implementation Details

This chapter primarily deals with the underlying architecture of the distributed rendering program. It explains how data and render nodes are synchronized to counteract cyber sickness. An additional overview is given to implemented renderers and resources, and their dependencies which also depict principal data flow. The subsequent section explains crucial details and stumbling blocks for efficient CUDA kernel implementations distributing light. Besides the CUDA execution model, it also explains the CUDA memory hierarchy and how to use it for efficient propagation. The kernel to propagate light in the virtual scene is implemented in the other frameworks OpenGL and OpenCL as well, both of wich deliver different performance results. Differences between these kernels are briefly explained to allow better understanding of the subsequent performance comparisons. Finally, this chapter concludes with performance measurements and results, combined with important implementation details about the measurement process.

4.1 CAVE Cluster

The application is designed to run in a multi-display setup, although each logical node is limited to a single render window. This limitation may be easily lifted to support a wider range of setups which is however not required for the application to work in the CAVE cluster at the Leibniz Supercomputing Centre (LRZ) (figure 4.1). This 5-sided CAVE consists of 10 render nodes and 2 master nodes, each render node projecting a half side at a resolution of 1920x1080. Here only one master node is active controlling the render nodes uni-directionally[1] and render nodes do not communicate with each other. Due to high display resolutions and the requirement for real time refresh rates, distributed rendering in combination with incremental render parameter updates is inevitable.

Each node is equipped with a NVIDIA Quadro allowing an efficient hardware lock between all GPUs in this cluster to ensure synchronization. At the time of writing, render node 2 serves as a clock server for this lock between the remaining 9 render nodes.

Rendering in the CAVE is facilitated by the Equalizer[2] framework which setups a synchronized renderer cluster depending on a configuration file. The choice for the rasterizer is OpenGL since the Equalizer framework is based on it. Since the application is almost entirely replicated on all nodes, no OpenGL rendering commands are transmitted through the network. Data is not shared between render nodes since they do not communicate with each other.

[1] Note that the Equalizer framework supports more dedicated communication modes, but for the application of this thesis a simpler scheme suffices.

[2] See `http://www.equalizergraphics.com/`.

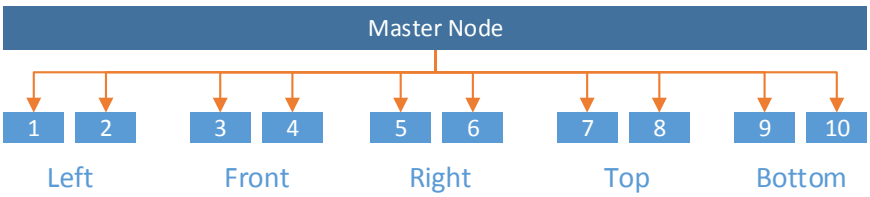


Figure 4.1: 5-sided CAVE cluster topology. 1 master node (dark blue) synchronizes 10 render nodes (light blue, numbered) over a LAN (orange arrows). Each render node projects a half side of the 5-sided CAVE. Principal communication direction is shown by arrows.

4.2 Architecture

The implementation is build on top of the Equalizer framework to synchronize rendering between all render nodes in the CAVE cluster (figure 4.2). The application further employs plain OpenGL Strict 4.2 C API, GLSL 4.2 shaders, OpenCL 1.1 C and CUDA 6.0 kernels propagating light through the lattice. The C++ Standard Template Library (STL) is used as much as possible saving implementation effort. The modular architecture can be easily extended by new renderers, render pipelines and scenes. The helper module `clx` facilitates creating OpenCL contexts and programs by using factories. Loading, compiling and linking shaders into GLSL programs is made significantly easier with the modules `gl` and `glsl` featuring detailed error handling. The latter helps debugging shaders during development. Here a factory is initialized with a shader directory and known shader extensions. Already loaded shaders can be reused to save GPU resources. It helps to retrieve uniform locations of GLSL programs as well. The module `core::input` contains abstract input handlers and various input sources, e.g. keyboard, mouse, VRPN analog device and VRPN button. These input sources receive and deliver input to a registered application specific input handler. In this application, input handlers send appropriate parameter updates to a synchronized distributed data structure. The module `core:xeq` implements Equalizer configs, nodes, windows, pipes, channels and synchronized data which can be reused for similar programs with this architecture. It allows more fine grained control than `seq` (Simple Equalizer) while hiding technical details e.g. to properly setup data synchronization, input handling, render pipeline setup and render pipeline execution. The application runs with CUDA 6.0 and GLEW 1.10 to support latest OpenGL features for performance optimization. By using CUDA, a CUDA capable graphics device must be present on each node in the cluster.

Frames are synchronized by calling finishFrame in the master node. This triggers glFlush and glFinish in all render nodes, therefore sending all GL commands to the GPU as fast as possible, and blocking the host until all GL commands are terminated. In practice, this procedure safely synchronizes both host render thread and device render thread. Usually, these threads are asynchronous with a latency of about 2 frames for performance reasons. In the asynchronous case, there is no guarantee that each GPU in the CAVE cluster sends the same frame at the same time to the display adapter, because rasterization time also depends on camera perspective, i.e. view transformation.

Application, Initialization, Configuration Files

The application instance is run by both master and render node, but the behavior differs depending on command line arguments. On the master node, the application object initializes

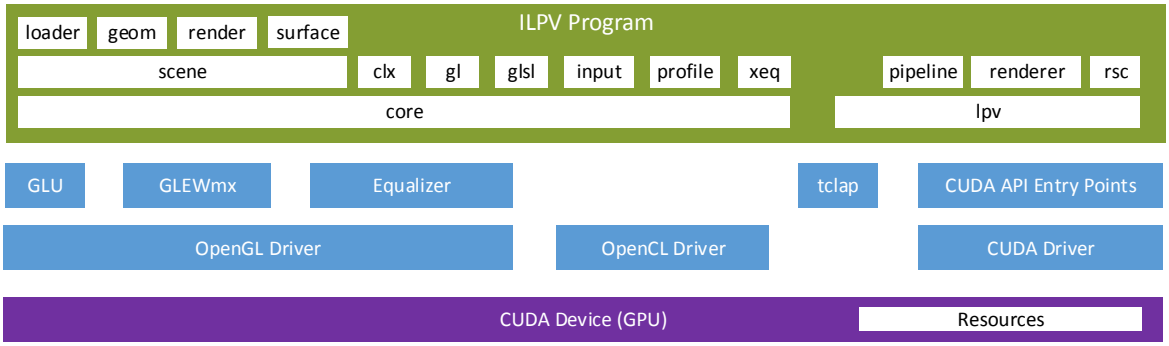


Figure 4.2: The application contains the modules `lpv::renderer`, `lpv::pipeline`, `core::scene`, `core::input`, `core::xeq` (extended Equalizer), `core::clx` (extended OpenCL), `core::gl` (extended OpenGL). It builds on top of Equalizer 1.7, OpenGL 4.2 Strict rasterizer, CUDA 6.0 and a CUDA capable device holding created resources for rendering, e.g. buffers, textures, shaders and programs. Note the Equalizer build contains VRPN while GLEW 1.10 is linked to as a separate library.

Equalizer with a path to an Equalizer configuration file[3]. It subsequently calls an Equalizer configuration object to setup all render nodes having windows, channels and pipes (figure 4.3). It also creates a constant distributed object having all initial data needed for the render nodes to start working, e.g. a global absolute path to a scene configuration file[4]. This file contains relative paths to models, light sources and further configuration parameters for all renderers and pipelines to be used during initialization. Some parameters are e.g. the default number of propagation steps, the lattice size and whether to render incrementally. Parameters are stored in an extended simple INI file format.

For input handling, a separate file[5] stores configuration parameters for each input sources to be used. For simplicity, each INI section is associated by a named input source. Thus each input source type can be activated only once during runtime. This limitation suffices for the implementation requirements.

During rendering, the application object starts and finishes a frame ① by calling the underlying Equalizer configuration for the CAVE cluster. This configuration instance then calls all known render nodes in the cluster ②. Each render node updates distributed objects to their head version ③ and finally starts the currently active render pipeline ⑦. Note that finishing a frame is not shown in figure 4.3 since it works similarly. Note however that the next frame is only started after all render nodes signaled the master node of having finished the frame which is sent to the display adapter. This behavior ensures synchronization, i.e. no render node lags at least one frame behind. With failed synchronization, the observer's left and right eye may see renderings using a different scene or changed parameters, finally reducing immersion.

Input Handling, Updating Configuration

Here, input handling is centrally managed by the master node. Depending on configuration, multiple input sources can be supported, e.g. a keyboard, a mouse, head tracking, analog

[3]This path is given to Equalizer by using `--eq-config` ⟨`path`⟩.

[4]The scene configuration path is given by the argument `-s` ⟨`path`⟩. It simplifies program startup since the render pipeline uses many parameters. It is also used for scene models, lights and paths to shader and kernel directories.

[5]The argument `-c` ⟨`path`⟩ defines the path to the input config file.

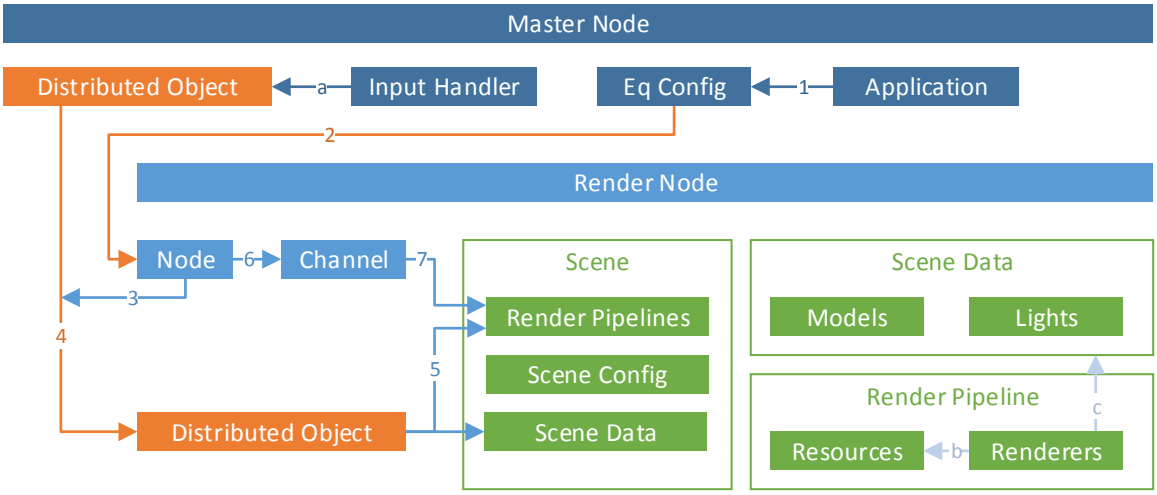


Figure 4.3: Each render node (light blue) holds a synchronized copy of all scene data, configuration data and render pipelines with render instances (green). The master node (dark blue) is responsible for starting and finishing frames ①, and for handling input ⓐ. Principal call sequence and data flow to start rendering a frame is shown as numbers. Data is also transfered over LAN (orange). Renderer instances generally depend on resources ⓑ and scene data ⓒ.

devices and VRPN buttons.

If enabled, head and wand input sources get their data from the VRPN library, abstracting input devices. The input source object further abstracts the device and basically directs input to a single input handler object managing triggers for input events. This input handler incrementally sends modified configuration values to a synchronized distributed object supporting version numbers ⓐ. On each render node, the node synchronizes this object to its head version ③ and all modified data are transfered over the LAN ④. The updated parameters are sent to the corresponding renderers of the currently active render pipeline ⑤. Note that sending these parameters to render nodes is synchronized in the CAVE cluster, since version numbers are employed and all render node's synchronized objects are updated to their head version before each frame. This comes at a cost of about 0.5 ms on a local node, and approx. 11 ms in the CAVE cluster showing a network issue. This latency makes up a fourth of total stereoscopic rendering frame time including render node synchronization, given a lattice resolution 64^3, 8 iterations, 1M triangles. In comparison, a virtual 2 node cluster in a single physical node yields a synchronization latency between 0.5ms and 1ms.

Scene, Data, Resources

Each render node has a scene being a central object containing scene data, a parsed scene configuration object and a list of available render pipelines. This scene configuration is used during initialization and for scene reloading. The implemented render pipeline also stores render resources, e.g. a lattice descriptor, textures and framebuffers. Resources are shared objects between renderer instances. They simplify architecture, speed up compilation and ensure that all renderers use the same set of data, particularly in case of data modifications. Each such resource is initialized using parameters from the scene configuration file.

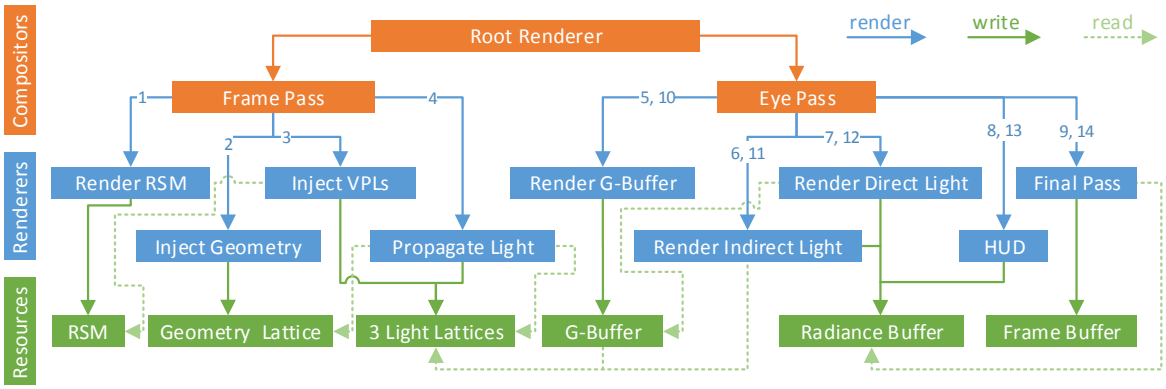


Figure 4.4: The slightly abstracted renderer graph with render passes as compositors (orange), calling attached renderer instances (blue) in-order as indicated with the blue numbers (top to bottom, left to right). The renderers store (green arrows) and read (green dotted arrows) data in and from resources (green), respectively. The modular architecture is easily extendable and allows efficient computation.

Frame Pass, Eye Pass, Renderer Graph

Before initializing all involved renderers in the pipeline, an efficient renderer graph is constructed once (figure 4.4). This is done by first linking all created renderer objects to resources as needed, and then by inserting them into order-aware compositors. By using compositors, calling init methods and render methods greatly improves code readability, at almost no additional cost by using cache coherent STL vectors.

All renderers are finally combined in a root renderer of this Directed Acyclic Graph (DAG). However, this root renderer is not used during runtime to distinguish between a view-independent and a view-dependent rendering pass, i.e. between frame pass and eye pass respectively. During the frame pass, RSMs are rendered, both VPLs and scene geometry are injected and finally light is propagated. This is done once per frame. Subsequently, the eye pass is called for the left and right eye, thus rendering view-dependent indirect and direct illumination, including a HUD. The eye pass finishes by applying tone mapping and gamma correction in the final pass.

4.3 Propagation in CUDA

Propagation of indirect illumination has been implemented in OpenCL, GLSL Compute Shader as well as in CUDA during this thesis. It turns out that CUDA yields best performance because it allows efficient use of memory hierarchy. In particular, the CUDA driver distinguishes between registers and shared memory, both on-chip, and between L2 cache, read-only texture cache, constant and global memory, which are all off-chip on the GPU device. Concerning OpenCL and GLSL, they do not support low-latency shared memory between executing kernel threads. In CUDA, reading from shared memory can be as fast as register loads, requiring a few clock cycles only. As shared memory is on-chip, it offers high bandwidth being particularly important for memory bandwidth bound applications. However this memory type is quite limited to about 32Kb or 64Kb, depending on hardware architecture generation.

For example, the Fermi architecture[6] supports up to 48Kb shared memory per block. In the CUDA execution model, the grid (device) consists of many thread blocks (multiprocessors). Each block has several threads (scalar processor) to be executed in parallel if possible.

Each Multiprocessor (MP) has its own set of registers and shared memory (48Kb). Being highly important, the latter enables thread coorporation within a single block. As each thread loads neighbor cells, global memory fetches can be significantly reduced. Without using shared memory, the kernel is memory bandwidth bound because in this case each cell data is loaded 26 times which forces thread stalls and memory pipeline overuse. This non-coalesced global memory access can be further avoided with shared memory, although greatly depending on load patterns within the kernel. Registers hold data for local kernel variables, which is not equal to local memory in CUDA being stored off-chip. Global and local memory both suffer from the highest load latency for device memory[7], but feature the largest memory since fast memory is expensive. By distributing data to different memory types, effective memory bandwidth can be increased, further improving performance.

Kernel Optimization

In the CUDA execution model, each thread propagates light from 26 neighbors to its corresponding cell, i.e. instead of scattering a gathering approach is employed which works well with shared memory and fits better to current GPU architectures since it saves expensive memory stores. These threads are further grouped into uniformly sized blocks. Before actual propagation, each block thread loads 2 or more cell data into the block's shared memory. This is required because the bordering block threads may read data being out-of-block by one cell during propagation due to the 26 neighbor scheme. In detail, the thread (x, y, z) does not load the cell (x, y, z) into shared memory[8]. Instead each block thread is flattened to a number i, from which a cell correspondence (x', y', z') is derived to account for the extra one cell border around the block (figure 4.5). In particular each thread trivially loads multiple cells so that the largest mapped cell (x', y', z') is loaded. These data are preloaded in parallel between block threads, although each thread loads data sequentially. Effectively data is loaded only once from slow global memory into fast shared memory within each block. However, overlapping "cells" between blocks are loaded between 4 and 8 times. Therefore the larger the block size, the more efficient the kernel execution. But the block size is limited to 8^3 due to limited shared memory of current GPU architectures.

In addition, the two 3D arrays[9] making up shared memory are incremented by one in their z-direction which however remains unused during propagation. It turns out that this additional storage reduces shared memory bank conflicts due to less non-coalesced accesses during propagation. In detail, a bank conflict occurs if different threads request loads to shared memory addresses within the same bank since the memory can only service one address per bank per cycle. By using memory padding, memory transactions seem to be better aligned. Note also that performance is only improved if the array is incremented in its

[6] GPUs with Fermi architecture are GeForce 400, GeForce 500, Quadro 6000. The latter as well as GeForce 470 are used for performance measures.

[7] The term "device" is often associated with the GPU, where "host" often corresponds to CPU, main DRAM

[8] Although this approach would make loading data into shared memory simpler, it also forces the extra threads around the original block to be prematurely terminated. Because they remain unused, performance drops significantly. For example, a block size of 8^3 would stall then $8^4 - 8^3 = 3584$ threads, which is about 87.5% stalls.

[9] These arrays hold data for one color component and the blocking SH coefficients

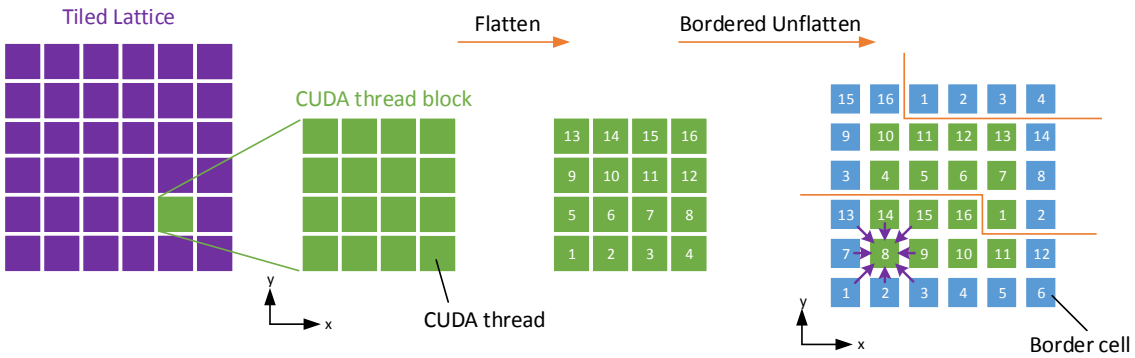


Figure 4.5: CUDA threads (blue rectangles) are flattened and subsequently assigned multiple shared memory cells (x', y', z'), corresponding to a thread block with one cell border (yellow). This approach allows fair work distribution among threads, i.e. loading cells from slow global memory to fast on-chip shared memory. Note the z-dimension has been omitted for simplicity.

z-direction only, i.e. x- or y-direction has not changed performance in own experiments. A more detailed explanation is beyond the scope of this thesis[10].

Finally, after all data is loaded into block's shared memory, all threads are synchronized ensuring defined computation in subsequent code. Here the synchronization overhead is kept small if loads are fairly distributed among threads which is done by the flattening approach (listing 5.2). After synchronization, light is gathered from 26 neighbor cells stored in shared memory in a for loop[11], and finally stored in global memory for the corresponding destination 3D CUDA surface which maps the OpenGL 3D texture resource.

Threads and OpenGL Interoperability

Propagating light in CUDA involves at least two threads, a render thread on the host and CUDA kernel streams on the device (figure 4.6). In fact, the kernels employ 3 streams, each corresponding to a color component. Before executing kernels, previous rendering to OpenGL resources must be finished, e.g. by calling glFlush and glFinish. Otherwise acquiring and using resources in CUDA results in undefined behavior by specification. GL resources are acquired and mapped to CUDA 3D surfaces and textures with underlying CUDA 3D arrays. This steps consumes some nanoseconds only being neglectable compared to kernel execution. The incremental scheme which practically merges two lattices with multiplicative factors is executed in a separate CUDA kernel asynchronously. By calling an external C function written in CUDA from the render thread, this execution is started 3 times with separate streams. The host render thread then calls another external C function hiding initialization and call details for the kernel. For example, it precomputes variables remaining constant during propagation, it determines the block and grid size, it binds surfaces and textures to 3D arrays and it starts kernel execution. This is repeated for the number of propagation steps. Note that during the first step, blocking is disabled. Here kernels are executed in separate streams as well. Depending on hardware and particularly on shared memory size, kernels may be executed in parallel which is only possible if different streams are employed.

[10] See CUDA 6.0 manual for more details

[11] The for loops in CUDA kernels are ensured to be unrolled by the assembler, i.e. a pragma directive is used. It requires only few more registers, but ensures optimization.

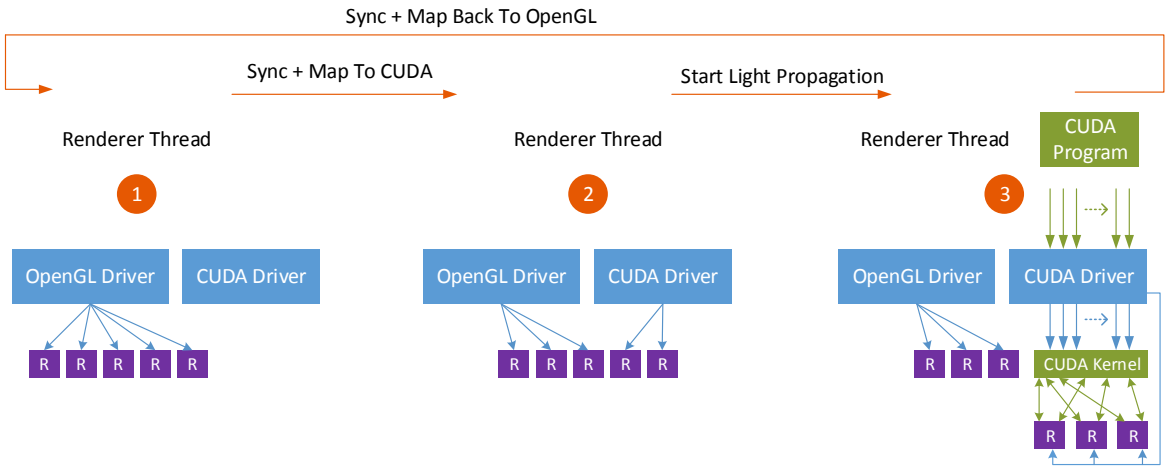


Figure 4.6: Abstract call sequence when propagating light in CUDA. Before propagation, device resources need to be mapped from OpenGL (state 1) to CUDA context with synchronization (state 2). Afterwards these resources may not be used by OpenGL. In state 3, the renderer thread calls C external functions compiled by NVCC to start the kernel for lattice merging due to incremental scheme, and the kernel to propagate light for each spectrum component. After propagation, acquired device resources are mapped back to OpenGL for further processing (state 1).

Since these kernels are run asynchronously, the host render thread needs to wait for them to finish, so that CUDA can safely unmap resources.

4.4 Propagation in OpenCL

Propagating light using an OpenCL kernel is currently limited by the absence of both loads and stores concerning acquired 3D textures in NVIDIA drivers. Therefore the driver has to copy data between 3D OpenGL textures and OpenCL arrays, introducing a latency for large lattices. Here data can be copied directly within the device or via the host. Performance numbers are provided only for the latter approach. In contrast to CUDA, OpenCL does not offer explicit shared memory management, leaving device memory usage to the driver. These circumstances are reflected in the provided performance numbers.

4.5 Propagation in OpenGL

In contrast to OpenCL, OpenGL natively supports 3D textures. In addition, this approach dispenses with synchronization before and after propagation since operations stay within the same driver. Here shaders propagate light within the lattice.

The Compute Shader (CS) is quite similar to either OpenCL or CUDA kernel, except that light is propagated for 3 RGB components at once to employ optimized vector arithmetics (listing 5.3). Image load and store functions provide data while no shared memory is utilized explicitly. A local size $(4, 4, 4)$ showed best performance.

Almost the same code within a geometry shader significantly drops performance due to less GPU resources being given to this shader stage. In particular, the geometry shader provides no comparable concept of block size configuration compared to OpenCL and CUDA. This

shader implements image loads and stores as well, no vertices are emitted and no pixels are drawn into the framebuffer. This kernel is named ILS in this thesis.

In case the geometry shader emits gathered cell radiances to the pixel shader to additively blend with a 3D texture framebuffer, performance slightly increases. Instead of image loads, texelFetches provide cell data. This kernel is named GLSL in this thesis.

4.6 Performance

Comparable performance measurements require not only both high resolution measurements and GPU device specification, but also GPU utilization, voltage levels and frequencies in a timeline. Besides, measuring time intervals may not stall either CPU or GPU render thread due to the asynchronous execution of OpenGL commands. Thus simply taking CPU timestamps yields unreliable results, e.g. when measuring draw calls.

Reliable Profiling

The presented performance numbers are based on two high resolution profilers: an asynchronous OpenGL profiler and a non-blocking host profiler using timestamps. All intervals are measured in nanoseconds, further including minimum, mean and maximum times.

The asynchronous OpenGL profiler employs OpenGL timer queries in a shifted round-robin style with a dynamic latency of at most 3 frames. This ensures that taking measurements does not stall the OpenGL render pipeline, therefore providing reliable results concerning OpenGL API calls. In practice, when calling GL API functions, they are enqueued to be sent to the GL, without blocking the host. This behavior however depends on the underlying graphics driver, possibly varying between versions, operating systems, and between desktop and console systems [Hil12, p. 353]. Furthermore, it is not guarenteed to having only one command queue. Depending on the device capabilities and on the driver, OpenGL API calls may even overlap and thus may be executed in parallel [Hil12, p. 353].

The second profiler measures time intervals on the CPU by using non-blocking calls to glGet with GL_TIMESTAMP, returning the current system timestamp in nanoseconds. This profiler is used to measure the time of propagation after calling clFinish, blocking the caller. Besides, both total time per frame and time between frames are also measured to investigate synchronization times and latencies concerning the Equalizer framework. These timings greatly influenced the optimization process of cluster synchronization and of updating shared objects.

In addition, the profiler should keep track of the current Performance State (P-State) including GPU utilization [Dim12]. Depending on the device driver, the driver API exposes a number of P-States, where the lowest state corresponds to idle state, and the highest state is entered when maximum performance is needed. Each state usually uses different voltage levels, device frequencies and memory-clock frequencies. This behavior not only reduces power consumption and produced noise, it also affects results from profiling graphics calls. Moreover, the driver may aggressively change the P-State, therefore P-State tracking is required. Unfortunately, NVAPI being used to query the current P-State is not available for Linux at the time of writing. Thus P-State tracking is not accounted for in the profiling results.

Besides, performance can be further increased by reducing calls to the driver, thus reducing state changes. However, these possible improvements should be neglectable in contrast to

						Frame Rate		
	G-Buffer	RSM	SI	FP	Direct Light	Local	C1	C2
Crytek Sponza	1.3	0.3	—	0.4	0.6	96.9	29.6	14.7
Box	2.8	1.7	—	0.4	0.6	74.7	29.5	14.8
Box Illum	2.8	1.7	12.8	0.4	0.6	38.4	13.9	14.2
San Miguel	8.1	7.0	—	0.4	0.6	42.6	14.6	14.7

Table 4.1: Performance measurements of render phases for selected scenes, including surface injection (SI) and final pass (FP). Resolutions: 1920x1080 (framebuffer), 256^2 (RSM), 32^3 (lattice). Propagation employs ILPV and CUDA with 8 steps. Final refresh rates are included for local non-stereoscopic rendering (Local), and low (C1) and high (C2) resolution stereoscopic rendering in the 10-node CAVE cluster. C1 employs the same resolution as Local, while C2 employs 512^2 (RSM) and 64^3 (lattice).

the algorithms used, e.g. to propagate light, to share GL objects between contexts or to efficiently transfer memory asynchronously.

Performance Results

Finally performance is reliably measured for different scenes, important rendering phases and for 2 Equalizer configurations (table 4.1). The local node configuration features a NVIDIA GTX 470 with Core2Quad 6600 (2.6 GHz). The CAVE configuration is made up of 10 render nodes, each having a NVIDIA Quadro 6000 with Intel XEON 3.5 GHz. Concerning the G-Buffer performance times, each scene model is stored in a single static Vertex Buffer Object (VBO) and Index Buffer Object (IBO), filled with a single precision floating point based indexed face-set being compressed with instancing. Here no packing is employed.

The CAVE requires a long synchronization chain between display projectors, head and wand tracking devices, device input server, shutter glasses, GPU devices, masternode-to-rendernode relation, frame-to-frame relation. Synchronization has to guarentee that for each observer's eye the correct projection on each display is displayed at the same time, while left and right eye alternate. In the CAVE at the LRZ, each frame is limited to 30 Hz in stereoscopic setups which is represented by column C1 in this table. In case the local refresh rate drops below 60 Hz, frame rate in the CAVE drops below 30 Hz. Investigating NVIDIA SMI revieled a GPU-utilization of approximately 68% for medium sized scenes (Box, 64^3, render node), and 92% for large sized scenes (San Miguel, 64^3, render node) while the master node showed generally less utilization of about 25% relative to render node utilization. As such, refresh rates for rendering complex scenes in high resolutions suffer less in CAVE installations than at local desktops in case of scene-space based approaches, as indicated by column C2. This circumstance becomes particularly apparent for the large scene "San Miguel" with 5.5M triangles which renders at the same frame rate than "Crytek Sponza" with 262K triangles for a 64^3 lattice resolution. Finally total GPU memory usage of the rendering process is 806MiB for "San Miguel" and 656MiB for "Box" (64^3).

Table 4.2 lists propagation performance for both different lattice resolutions and implementations. As discussed, CUDA clearly outperforms OpenCL and GLSL. Finally, table 4.3 shows important scene properties influencing performance or rendering quality.

		Propagation					
	Injection	CUDA	OpenCL	CS	ILS	GLSL	Indirect Light
32^3	0.7	2.7	5.8	6.1	19.9	18.7	2.7
64^3	0.5	18.2	66.7	49.5	184.1	142.4	2.7
128^3	0.4	140.9	598.5	407.6	—	—	2.7

Table 4.2: Propagation performance for different lattice resolutions with ILPV scheme. Measurements are generally independent on scene, although minor differences exist. They are based on GTX470. Numbers already include synchronization and buffer mapping. Propagation is measured in CUDA (fastest), OpenCL, OpenGL Compute Shader (CS), OpenGL Image Load and Stores (ILS) and old OpenGL GLSL shader without image stores. Both injection and indirect light evaluation are independent of the propagation implementation here. Resolutions: 1920x1080 (framebuffer), 256^2 (RSM).

			Surface Lights	
Scene	Triangles	Dimension	Triangles	Flux
Crytek Sponza	262.2K	371x156x229	—	—
Box	1.08M	11x10x10.5	—	—
Box Illum	1.08M	11x10x10.5	2.61M	120
San Miguel	5.54M	68x15x27	—	—

Table 4.3: Important scene properties which significantly influence either performance or rendering quality.

5 Conclusions and Future Work

This thesis presents the application of LPV rendering for the first time to CAVE installations, as far as known by the author. It also derives a new rendering scheme ILPV which reuses light distributions from previous frames both to save propagation steps and to increase rendering quality.

Besides, the transfer vectors are computed with stochastic rejection sampling, combined with stochastic light distribution sampling while solving a minimization problem to approximate the law of energy conservation during propagation. The geometry lattice is ensured to have no holes by stochastic uniform sample distribution for each mesh triangle. Here the sample count depends on triangle area and lattice configuration. Arbitrary surface lights are rendered as well, although improvements should be researched.

The presented rendering architecture is both efficient and modular, making further development easier. It clearly separates resources from renderers which initialze themselves given a loaded configuration file. Propagation has been implemented in CUDA, OpenCL and OpenGL framework. The CUDA kernel requires approximately 3ms for 8 propagation steps and 32^3 lattice on Fermi hardware including management overhead. Final tone mapping and gamma correction convert HDR images to LDR pixel values to be properly displayed depending on display color space.

Performance measurements and results clearly indicate the significant advantage of scene-space based GI approaches in the context of stereoscopic rendering, e.g. LPV and ILPV compared to camera view dependent IR.

Although ILPV increases propagation distance, performance is still primarily limited by lattice resolution.

Concerning propagation for particular large light lattices, load-balancing light propagation on all render nodes and synchronizing their results would be more efficient. This approach prevents each render node from distributing all light through the entire scene in a duplicate manner. Instead, each node might possibly propagate light for a scene subset which however requires non-trivial merging and delicate handling of overlapping regions. More investigations are required as to which solutions work best in a CAVE cluster, weighted by implementation complexity.

Currently LPV approaches are inherintly performance-wise limited by both lattice resolution and propagation distance. Instead of interpolating between 3 different resolutions [KD10], a single non-uniform lattice data structure may mitigate these restrictions as well. This structure should be optimized for stereoscopic rendering to minimize structure updates between left and right eye off-axis projections. Similar to ideas in LOD algorithms, structure resolution should decrease with increasing distance from camera eyes. This non-uniform structure imposes a difficulty in case uniform propagation speed in world space is desired.

A substantially simpler improvement may use redundant OpenGL resources. For large lattices, e.g. 128^3 or larger, performance may be improved by duplicating some OpenGL resources for 2 or more frames to be used in a round robin style for propagation and rendering. This approach reduces CPU stalls when synchronizing between OpenGL and

CUDA, effectively improving refresh rate. This is especially useful if the application is not GPU bound, e.g. in CAVE applications which introduce a latency between frames for both synchronization and internode communication purposes.

Currently, rendering surface lights with ILPV is experimental. Here these lights approximated by triangle meshes are not utilized in direct illumination rendering. This limitation causes visible artefacts, seen as too dark light surfaces since only indirect illumination is rendered. In addition, visible light sources usually have a corona around their silhouette, further increasing rendering quality. Since performance is quite limited in CAVE installations compared to single node real time rendering, this feature has also been kept back for future work.

Regarding user interaction in CAVE environments, movement velocity is determined by wand-body distance. However for simulating walking, this velocity also depends upon virtual surface relief. Further research should be done for finding a mathematical relationship between velocity and both ethnicity and age of the human observer.

Finally, further investigation can be carried out to which extend the proposed ILPV approach works well with an additional accumulation buffer, particular in combination with work contributed by [BSA12; Bør+11; Fat09; KD10]. As shown in this thesis, ILPV significantly increases rendering quality at almost no additional expense. A few propagation steps already yield very good results. When applied to these works, ILPV can greatly help to improve rendering.

User Interaction

1 Overview

While the user is inside the CAVE, a Heads Up Display (HUD) is rendered on top of the final image helping the user both to navigate in the scene and to change configuration settings of the renderers, e.g. the number of propagation steps or the rendering quality.

2 Navigation

Besides head tracking, a wand is used both to navigate in the scene and to trigger functions. This is particularly required for large scenes, and to keep immersion and user experience high.

The wand used for this thesis features 4 buttons and 2 analog axes (figure 1). These buttons shall be named as seen from the user: left Ⓛ, middle Ⓜ, right Ⓡ, control button Ⓒ, left to right axis Ⓧ, and bottom to top axis Ⓨ. Due to limited buttons and to having more configuration values than buttons, one button Ⓡ changes the input layout. To simplify interaction, no long press buttons are implemented. In addition, no button combinations are supported since the input layout should be simple enough for easy memorization.

In general, pressing Ⓡ switches to the next higher input layout in a round robin style. Defined input layouts are:

1. Scene navigation
2. Spotlight movement, its temperature and intensity
3. Rendering settings, e.g. propagations, absorption and scattering

Layout 1: Scene Navigation

As soon as the visualized scene is larger than the CAVE, the user might actually need to navigate through the experienced VR. Being a limitation of CAVEs, the user can only walk within a small range up to the walls. However in correct implementations the user does not perceive these walls, ultimately colliding with them. Therefore the user needs a means to move inside the VR. Here a wand is employed.

By pressing Ⓒ, the CAVE's world position moves along the current wand's view direction. The movement velocity v_{cave} is non-linearly determined by the distance d_{wh} between wand and head being projected to the wand's height in CAVE space. Moreover to simulate natural walking, ${v_{\text{cave}}}^0 = 1.4ms^{-1}$ [Bro+06] for a distance of $d_{wh} = 0.4m$ which roughly approximates a lower arm length for adults. Velocity finally is

$$v_{\text{cave}}(d_{wh}) = d_{wh}^{2.5} \cdot 0.4^{-2.5} \cdot 1.4 \quad [ms^{-1}]$$

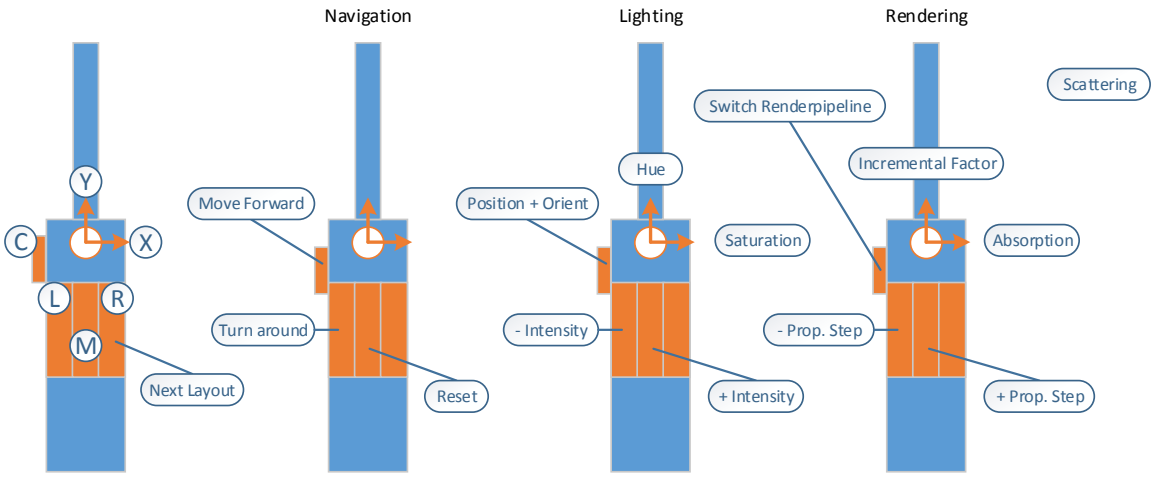


Figure 1: The wand used for this thesis has 4 buttons Ⓛ Ⓜ Ⓡ Ⓒ and 2 analog axes Ⓧ Ⓨ within $[-1, 1]$. Depending on input layout, these buttons are used to move and orient both the CAVE and the spotlight in the scene, to modify lighting, and to change rendering settings, e.g. number of propagations, absorption, scattering and incremental factor.

This however does not account for the user's age and ethnicity, and further refinement is required.

By moving the wand, the user can change the CAVE's orientation while holding Ⓛ pressed. When this drag action is initialized, both current CAVE's and wand's orientation are stored. Upon subsequent wand movements, a difference between current wand orientation and initial wand orientation is computed and used to update the CAVE's orientation, based upon its stored value. This drag action basically extends a 2-dimensional mouse drag action for CAVE contexts.

As a fallback, and for convenience, pressing Ⓜ resets both CAVE position and orientation to default values, e.g. the origin in world space.

Layout 2: Lighting

Dynamically lighting the scene requires e.g. a spotlight, a directional light, an environment light or a surface light. Surface reliefs can then be illuminated as requested by the user, further increasing realism and immersion. While pressing Ⓒ, the spotlight is matched to the wand's position and orientation in world space, thus acting like a flashlight in the user's hand.

Important characteristics of light sources are light intensity, emitted light spectrum, type of light, size of light's area and Field-of-View (FOV), among others. For a spotlight, the light's area is neglected. For large rooms, light intensity might have to be increased, which can be done through Ⓜ. Ⓛ lowers emitted light intensity. Changing the light spectrum is slightly more challenging since only two analog axis are left. For an RGB spectrum, full control needs 6 triggers to increase and decrease the components directly. Another less intuitive solution might derive the spectrum from 2 chromaticity coefficients, neglecting color brightness. This approach needs 4 triggers, e.g. Ⓧ and Ⓨ. However, it might not be intuitive to the user how these coefficients need to be changed to obtain a specific light spectrum. And another approach might even directly modify the spectrum components by

applying a more complex input mapping, e.g. Ⓛ button to select a component and Ⓧ to change its value. Using this approach, determining the active component is not apparent. An additional HUD primitive visualizing the currently set component might actually help the user, but it also makes the HUD more complex, although the full RGB color space is supported. A more intuitive solution may derive the spectrum from a given one dimensional light temperature modeling cool and warm lights. Being a projection to a lower dimension and an information reduction, this approach fails to represent the full RGB color space. Here green, turqouise and purple are particularly under represented. However changing the spectrum becomes straightforward. Instead of light temperature, the visible band of electromagnetic radiation provides particularly more colors. Similar the HSV color model provides an intuitive interaction to change both hue and saturation since these parameters are separately updated by the user. In comparison, the RGB color model makes configuration of fine grained pastel colors rather difficult. Here the third component, value, can be neglected since it is already indirectly represented by light intensity. Thus the remaining Ⓨ gradually changes hue, whereas Ⓧ modifies saturation.

Layout 3: Renderer Settings

Besides scene movement and lighting, the rendering configuration should also be changed by the user. Most importantly, the number of propagations are incremented and decremented by pressing Ⓜ and Ⓛ, respectively. Now, Ⓧ changes absorption, and scattering is modified through Ⓨ.

Depending on configuration, the second render pipeline computes GI in higher resolutions, e.g. for a 64^3 or 128^3 lattice. Ⓒ changes the active render pipeline. While the quality of both indirect lights and shadows are increased, frame rates drop, and thus rendering may be interactive. To keep immersion high, the render pipeline can be configured to only update the light distribution after primary light changes. This enables the user to investigate a large scene in high resolution, still at real time frame rates. Using this setting, rendering is switched back to the default render pipeline as soon as primary light changes, further ensuring real time frame rates. Note that this behavior can be easily customized using a configuration file.

3 HUD

The 3 different input layouts, explained in the previous section, complicates user interaction. Therefore, the user needs a visual indication of the current input layout.

For simplicity, visual indicators are rendered near the wand as geometric primitives having distinctive colors, e.g. arrows (navigation), cones (spotlight), cubes or torus (render settings). These primitives are colored according to the represented state. By default, all primitives belonging to an inactive input layout are gray. Once an input layout is chosen, its visual indicators are colored usually as white. An exception is the cone which always represents the spotlight's color. Its scale also depends on the spotlight's FOV to naturally show this type of light. This cone is also rendered at the actual spotlight's position and orientation, otherwise immersion suffers. These layout primitives are drawn on the wand's left side, thus the rendered HUD is wand-referenced, further increasing interactivity. On the wand's right side, the user perceives multiple render parameters. Since a bar visualizes each value, additional hints between wand and bar help the user to identify their meaning. Here 3-dimension letters

provide a clear hint. However care must be taken when positioning and orienting them. In detail, these parameters are absorption (A), scattering (S), incremental factor (I) and number of propagation steps (P) being normalized by lattice resolution. In addition, the visualized incremental factor is highlighted as white in case of incremental rendering. These elements are visually kept at its minimum preventing the user from being overwelmed by details. For a high degree of presence, no menus are embedded in 3 dimensional space.

Renderings

Figure 2: Absolute error of different configurations referenced to $(32, 0.0)$. Top left to right: $(4, 0.0, 0.128)$, $(4, 0.5, 0.067)$ and $(4, 0.8, 0.037)$. Bottom left to right: $(8, 0.0, 0.097)$, $(8, 0.5, 0.037)$ and $(8, 0.8, 0.026)$. These abbreviations correspond to (Iterations, λ_{incr}, MAE). Mean absolute error (MAE) shows better convergance of ILPV compared to LPV.

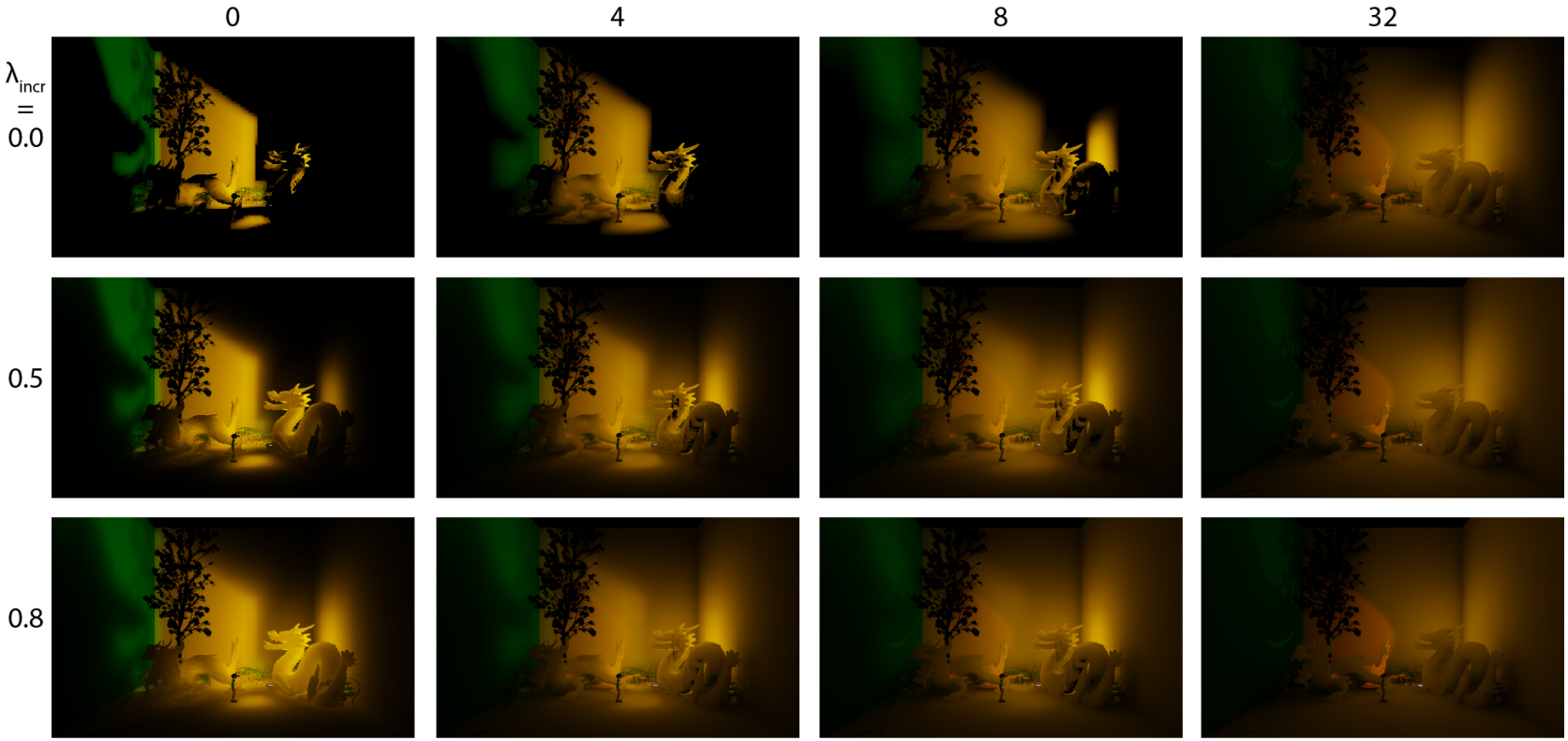


Figure 3: Comparison between both λ_{incr} (top to bottom: $0.0, 0.5, 0.9$) and propagation steps (left to right: 1, 4, 8, 32). Rendering with $(8, 0.9)$ is quite similar to $(32, 0.0)$. Remarkably $(1, 0.9)$ already provides good results, thus rendering with high lattice resolutions becomes efficient. In particular $\lambda_{\text{incr}} = 0.9$ still preserves indirect shadows. Note the indirect shadows between wall and dragon, and behind the human. Here the Box scene is rendered with 64^3 lattices and 512^2 RSM.

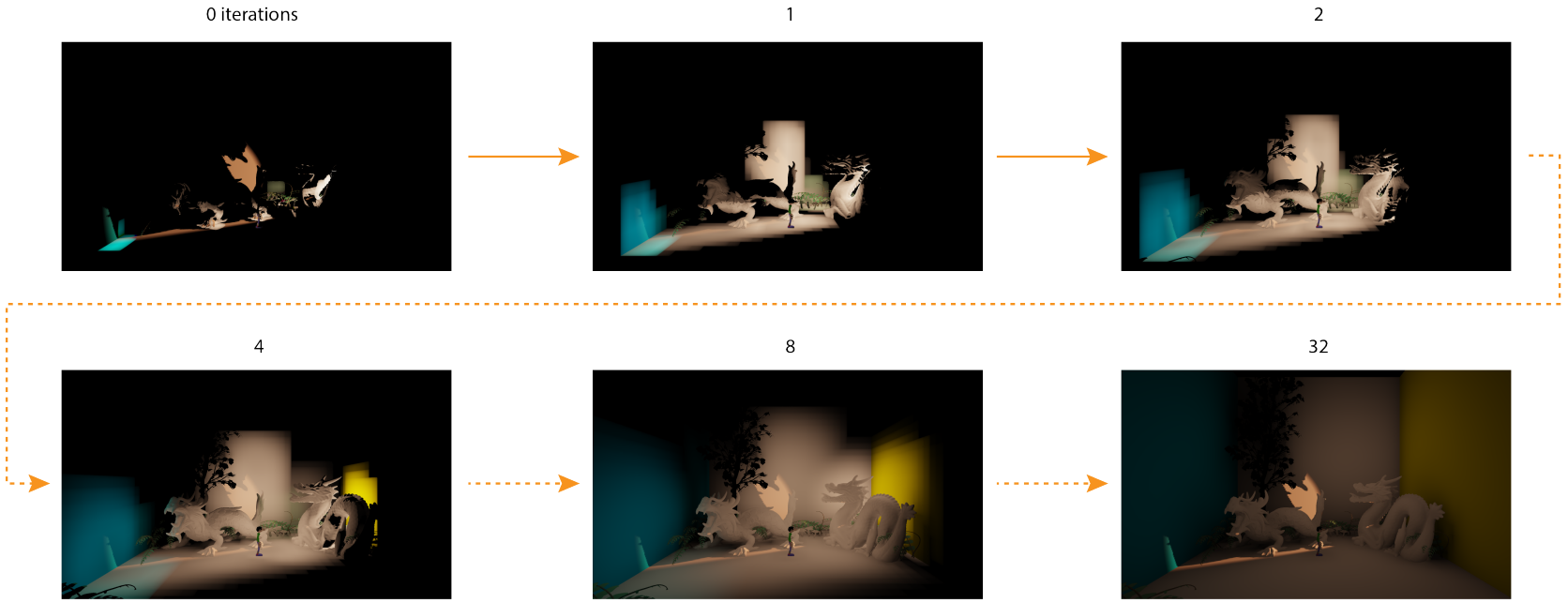


Figure 4: Comparison of propagation steps with the lattice resolution 32^3. This sequence visualizes how light is propagated in the virtual scene without the incremental approach. For a lattice 32^3, at least 32 iterations should be computed to approximate light distribution through the entire lattice.

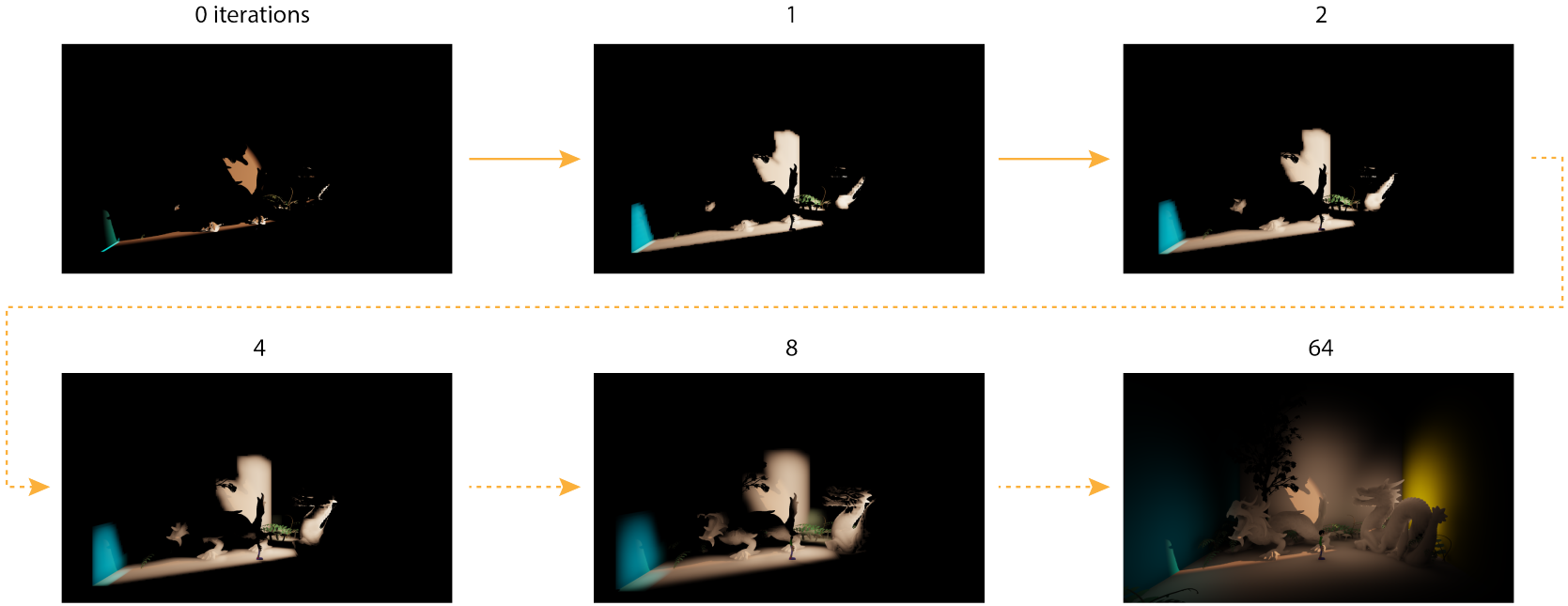


Figure 5: Comparison of propagation steps with the lattice resolution 64^3. This sequence visualizes how light is propagated in the virtual scene without the incremental approach. For a lattice 64^3, at least 64 iterations should be computed to approximate light distribution through the entire lattice which is not possible in real time. Compared to 32^3 after 8 iterations, 64^3 apparently visualizes the inherint limitation of LPV based approaches: scene space propagation distance is limited by resolution and number of iterations.

Figure 6: Visualization of applying the final pass consisting of tone mapping and gamma correction. Without tone mapping, out-of-gamut pixel values are sent to the display adapter, resulting in incorrectly white pixels.

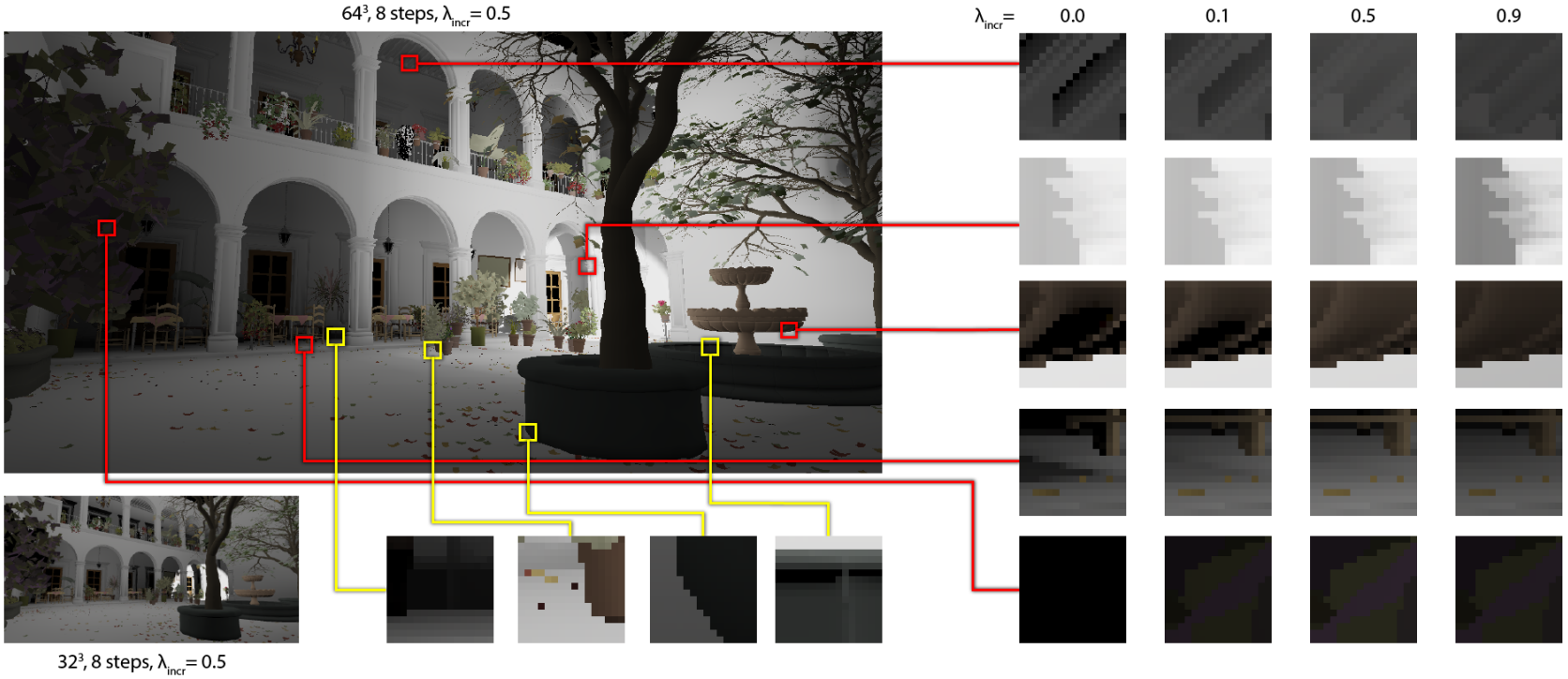


Figure 7: Rendering quality for variable λ_{incr}, based on the San Miguel scene (64^3, 8 propagations, no anti-aliasing, top left image). 5 16x16 samples (numbered from top to bottom) are magnified on the right, each comparing rendering quality for different λ_{incr} (left to right: $0.0, 0.1, 0.5, 0.9$). These samples show the superior quality for $\lambda_{\text{incr}} = 0.5$. Distant surfaces appear brighter due to increased propagation distance (sample 5), but near surfaces appear slightly darker (sample 2). Aliasing between lattice cells is mitigated (sample 4). In general, light distribution becomes smoother due to inherintly more propagation steps (sample 1, 3,4). Differences tend to vanish for 32^3 configurations (bottom left). Bottom middle images show characteristics and artefacts common among LPV approaches. Here the 64^3 geometry lattice fails to account indirect shadows for small objects (middle samples). Indirect shadows near the wall (left most sample) and self-shadowing near the fountain (right most sample) constitute of far less artefacts.

Code Snippets

Listing 5.1: CUDA kernel and entry functions to merge light lattice employed by the incremental approach.

```
struct MergeLatticesParams {
  float         factor_preframe;
  float         factor_curframe;
  cudaArray_t*  lattice_preframe;
  cudaArray_t*  lattice_curframe;
  unsigned int  lattice_resolution;
};

surface<void, cudaSurfaceType3D>        lattice_src;
surface<void, cudaSurfaceType3D>        lattice_dst;

cudaStream_t stream1, stream2, stream3;

/**
 * Merge two surfaces (lattices).
 *
 * This is used for incremental LPV
 */
__global__ void _mergeLatticeSingleSpectrum(float factor_src, float factor_dst
    )
{
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;
  int z = blockIdx.z*blockDim.z + threadIdx.z;

  float4 src, dst;
  int x_surf = x * sizeof(float4);

  surf3Dread(&src, lattice_src, x_surf, y, z, cudaBoundaryModeZero);
  surf3Dread(&dst, lattice_dst, x_surf, y, z, cudaBoundaryModeZero);

  src.x = __fmul_rd(src.x, factor_src);
  src.y = __fmul_rd(src.y, factor_src);
  src.z = __fmul_rd(src.z, factor_src);
  src.w = __fmul_rd(src.w, factor_src);
  dst.x = __fmaf_rd(dst.x, factor_dst, src.x);
  dst.y = __fmaf_rd(dst.y, factor_dst, src.y);
  dst.z = __fmaf_rd(dst.z, factor_dst, src.z);
  dst.w = __fmaf_rd(dst.w, factor_dst, src.w);

  surf3Dwrite(dst, lattice_dst, x_surf, y, z, cudaBoundaryModeZero);
}

extern "C"
cudaError_t lcrMergeLattices(MergeLatticesParams* params)
{
  cudaError_t err;
```

```
47   dim3 blockSize(4,4,4);
     dim3 gridSize(params->lattice_resolution / blockSize.x
49                 , params->lattice_resolution / blockSize.y
                   , params->lattice_resolution / blockSize.z);
51
     cudaChannelFormatDesc cd;
53   cudaGetChannelDesc(&cd, params->lattice_preframe[0]);

55   err = cudaBindSurfaceToArray(lattice_src, params->lattice_preframe[0], cd);
     cudaBindSurfaceToArray(lattice_dst, params->lattice_curframe[0], cd);
57   _mergeLatticeSingleSpectrum<<<gridSize, blockSize, 0, stream1>>>(params->
         factor_preframe
          , params->factor_curframe);
59
     cudaBindSurfaceToArray(lattice_src, params->lattice_preframe[1], cd);
61   cudaBindSurfaceToArray(lattice_dst, params->lattice_curframe[1], cd);
     _mergeLatticeSingleSpectrum<<<gridSize, blockSize, 0, stream2>>>(params->
         factor_preframe
63        , params->factor_curframe);

65   cudaBindSurfaceToArray(lattice_src, params->lattice_preframe[2], cd);
     cudaBindSurfaceToArray(lattice_dst, params->lattice_curframe[2], cd);
67   _mergeLatticeSingleSpectrum<<<gridSize, blockSize, 0, stream3>>>(params->
         factor_preframe
          , params->factor_curframe);
69
     return err;
71 }
```

Listing 5.2: CUDA kernel and entry functions to propagate light.

```
1  struct PropagationNeighborConstants {
     float4    transfer_vector;
3    int3      neighbor_dir;
     float     outscattering_fraction;
5  };

7  struct PropagationParams {
     float         scattering_coeff_norm;
9    float         absorption_coeff_norm;
     int           use_blocking;
11   cudaArray_t*  lattice_src;
     cudaArray_t*  lattice_dst;
13   cudaArray_t   lattice_geometry;
     unsigned int  lattice_resolution;
15   unsigned int  steps;
   };
17
   #define PROPAGATE_1PASS_BD_X    8
19 #define PROPAGATE_1PASS_BD_Y    8
   #define PROPAGATE_1PASS_BD_Z    8
21
   __device__ __constant__ PropagationNeighborConstants pnc[26];
23
   surface<void, cudaSurfaceType3D>        lattice_src;
25 surface<void, cudaSurfaceType3D>        lattice_dst;
   surface<void, cudaSurfaceType3D>        lattice_geometry;
27
```

```
texture<float4, cudaTextureType3D, cudaReadModeElementType>
    lattice_geometry_tex;

const surfaceReference*                         g_lattice_src_ref;
const surfaceReference*                         g_lattice_dst_ref;

cudaStream_t stream1, stream2, stream3;

/**
 * Propagate light in 1 pass while using shared memory and flattened mem
     loading
 *
 * Here each thread loads approx. 2 cells (block interior or block border). No
     threads are discarded, as in _propagateLightSingleSpectrum1Pass. This
     kernel should yield higher occupancy of SM (i.e. 2 blocks / SM is expected
     ).
 *
 * Consequently, the blocksize should not include a border when launching this
     kernel.
 *
 * Also see _propagateLightSingleSpectrum1Pass
 *
 * Experiments:
 * - using no smem for bcc dropped performance (loading via surface/texture
     cache) when calling tex3D inside the for-loop, 9 additional registers are
     used, resulting to 37 regs => limits 1 block/sm for 8^3 blocks (limit is
     34 regs).
 *   => performance dropped to 11ms, with 2 blocks/sm would not yield better
     than 2.7ms (tex3D for smem of bcc)
 */
__global__ void _propagateLightSingleSpectrum1PassFlattened(
  float inv_scattering_absorption_coeff_norm
  , float scattering_coeff_norm_sqrtpi_2
  , float use_blocking
  , int lattice_resolution)
{
  __shared__ float4 srcTile[PROPAGATE_1PASS_BD_X+2][PROPAGATE_1PASS_BD_Y+2][
      PROPAGATE_1PASS_BD_Z+3];
  __shared__ float4 bccTile[PROPAGATE_1PASS_BD_X+2][PROPAGATE_1PASS_BD_Y+2][
      PROPAGATE_1PASS_BD_Z+3];

  int threads = PROPAGATE_1PASS_BD_X * PROPAGATE_1PASS_BD_Y;
  int bxy = (PROPAGATE_1PASS_BD_X + 2) * (PROPAGATE_1PASS_BD_Y + 2);
  int bxyz = bxy * (PROPAGATE_1PASS_BD_Z + 2);

  // thread coords in bordered block space (first section)
  int i = threadIdx.z * threads + threadIdx.y * PROPAGATE_1PASS_BD_X +
      threadIdx.x;
  threads *= PROPAGATE_1PASS_BD_Z;

  #pragma unroll
  for (; i < bxyz; i += threads) {
    int iz = i / bxy;
    int iz_ = i % bxy;
    int iy = iz_ / (PROPAGATE_1PASS_BD_X + 2);
    int ix = iz_ % (PROPAGATE_1PASS_BD_X + 2);

```

```
    int x_gm = blockIdx.x * PROPAGATE_1PASS_BD_X + ix-1;
    int y_gm = blockIdx.y * PROPAGATE_1PASS_BD_Y + iy-1;
    int z_gm = blockIdx.z * PROPAGATE_1PASS_BD_Z + iz-1;
    int x_gm_surf = x_gm * sizeof(float4);

    if (x_gm < 0 || y_gm < 0 || z_gm < 0
        || x_gm >= lattice_resolution || y_gm >= lattice_resolution
        || z_gm >= lattice_resolution) {
      srcTile[ix][iy][iz] = make_float4(0,0,0,0);
      bccTile[ix][iy][iz] = make_float4(0,0,0,0);

    } else {
      surf3Dread(&srcTile[ix][iy][iz], lattice_src, x_gm_surf, y_gm, z_gm
                 , cudaBoundaryModeZero);
      bccTile[ix][iy][iz] = tex3D(lattice_geometry_tex, x_gm, y_gm, z_gm);
    }
  }

  int x_sm = threadIdx.x + 1;
  int y_sm = threadIdx.y + 1;
  int z_sm = threadIdx.z + 1;

  int x_gm = blockIdx.x * PROPAGATE_1PASS_BD_X + threadIdx.x;
  int y_gm = blockIdx.y * PROPAGATE_1PASS_BD_Y + threadIdx.y;
  int z_gm = blockIdx.z * PROPAGATE_1PASS_BD_Z + threadIdx.z;
  int x_gm_surf = x_gm * sizeof(float4);

  __syncthreads();

  float4 src, bcc;
  float4 dst = make_float4(0,0,0,0);

  float total_outscattered_energy;
  float blocking_factor;
  float propagate_factor;
  int x_src, y_src, z_src;

  #pragma unroll
  for (int d=0; d < 26; d++) {
    // gather from 26 neighbors => src_cell = dst_cell - neighbor_dir
    x_src = x_sm - pnc[d].neighbor_dir.x;
    y_src = y_sm - pnc[d].neighbor_dir.y;
    z_src = z_sm - pnc[d].neighbor_dir.z;

    src = srcTile[x_src][y_src][z_src];
    bcc = bccTile[x_src][y_src][z_src];

    // total energy = sqrt(PI) * 2 * src.x * scattering_coeff_norm
    total_outscattered_energy = __fmul_rd(src.x,
        scattering_coeff_norm_sqrtpi_2);

    // blocking = clamp(1.0 - <bcc|tv[d]> * blocking, 0.0, 1.0)
    blocking_factor = __fmul_rd(bcc.x, pnc[d].transfer_vector.x);
    blocking_factor = __fmaf_rd(bcc.y, pnc[d].transfer_vector.y,
        blocking_factor);
    blocking_factor = __fmaf_rd(bcc.z, pnc[d].transfer_vector.z,
        blocking_factor);
```

```
    blocking_factor = __fmaf_rd(bcc.w, pnc[d].transfer_vector.w,
        blocking_factor);
    blocking_factor = __fmul_rd(use_blocking, blocking_factor);
    blocking_factor = __fmaf_rd(-1.0f, blocking_factor, 1.0f);
    blocking_factor = __saturatef(blocking_factor);

    // propagate = max((1.0 - scattering_norm - absorbtion_norm)
    //              * <src|tv[d]> + total energy * outscattering_fracts[d]
    //             ) * blocking, 0.0f)
    propagate_factor = __fmul_rd(src.x, pnc[d].transfer_vector.x);
    propagate_factor = __fmaf_rd(src.y, pnc[d].transfer_vector.y,
        propagate_factor);
    propagate_factor = __fmaf_rd(src.z, pnc[d].transfer_vector.z,
        propagate_factor);
    propagate_factor = __fmaf_rd(src.w, pnc[d].transfer_vector.w,
        propagate_factor);
    propagate_factor = __fmul_rd(inv_scattering_absorption_coeff_norm,
        propagate_factor);
    propagate_factor = __fmaf_rd(total_outscattered_energy
                                 , pnc[d].outscattering_fraction,
                                    propagate_factor);
    propagate_factor = __fmul_rd(blocking_factor, propagate_factor);
    propagate_factor = fmax(propagate_factor, 0.0f);

    // transfer energy
    dst.x = __fmaf_rd(pnc[d].transfer_vector.x, propagate_factor, dst.x);
    dst.y = __fmaf_rd(pnc[d].transfer_vector.y, propagate_factor, dst.y);
    dst.z = __fmaf_rd(pnc[d].transfer_vector.z, propagate_factor, dst.z);
    dst.w = __fmaf_rd(pnc[d].transfer_vector.w, propagate_factor, dst.w);
  }

  surf3Dwrite(dst, lattice_dst, x_gm_surf, y_gm, z_gm, cudaBoundaryModeZero);
}

static int iDivUp(int a, int b)
{
  return (a % b != 0) ? (a / b + 1) : (a / b);
}

extern "C"
cudaError_t lcrPropagateLight1PassFlattened(PropagationParams* params)
{
  float inv_asn = 1.0f - params->scattering_coeff_norm
                  - params->absorption_coeff_norm;
  float sqrtpi_2 = 3.5449077f;
  float sqrtpi_2_scatt = sqrtpi_2 * params->scattering_coeff_norm;
  float use_blocking = 0.0f;
  int lr = int(params->lattice_resolution);

  cudaError_t err;
  dim3 blockSize(PROPAGATE_1PASS_BD_X, PROPAGATE_1PASS_BD_Y,
      PROPAGATE_1PASS_BD_Z);
  dim3 gridSize(iDivUp(params->lattice_resolution, blockSize.x)
                , iDivUp(params->lattice_resolution, blockSize.y)
                , iDivUp(params->lattice_resolution, blockSize.z));

  cudaChannelFormatDesc cd;
```

```
  cudaGetChannelDesc(&cd, params->lattice_src[0]);
177
  cudaArray_t* src = params->lattice_src;
179 cudaArray_t* dst = params->lattice_dst;
  cudaArray_t* swap;
181
  err = cudaBindTextureToArray(lattice_geometry_tex, params->lattice_geometry,
      cd);
183
  for (int i = 0; i < params->steps; i++) {
185   cudaBindSurfaceToArray(lattice_src, src[0], cd);
    cudaBindSurfaceToArray(lattice_dst, dst[0], cd);
187   _propagateLightSingleSpectrum1PassFlattened<<<gridSize, blockSize, 0,
        stream1>>>(inv_asn
        , sqrtpi_2_scatt, use_blocking, lr);
189
    cudaBindSurfaceToArray(lattice_src, src[1], cd);
191   cudaBindSurfaceToArray(lattice_dst, dst[1], cd);
    _propagateLightSingleSpectrum1PassFlattened<<<gridSize, blockSize, 0,
        stream2>>>(inv_asn
193       , sqrtpi_2_scatt, use_blocking, lr);

195   cudaBindSurfaceToArray(lattice_src, src[2], cd);
    cudaBindSurfaceToArray(lattice_dst, dst[2], cd);
197   _propagateLightSingleSpectrum1PassFlattened<<<gridSize, blockSize, 0,
        stream3>>>(inv_asn
        , sqrtpi_2_scatt, use_blocking, lr);
199
    use_blocking = 1.0f;
201   swap = src;
    src = dst;
203   dst = swap;
  }
205
  return err;
207 }
```

Listing 5.3: OpenGL compute shader to propagate light.

```
1 #version 420 core
  #extension GL_ARB_explicit_uniform_location : enable
3 #extension GL_ARB_compute_shader : require

5 layout(local_size_x = 4, local_size_y = 4, local_size_z = 4) in;

7 // Lattice configuration
  layout(binding = 0, rgba32f)  uniform readonly image3D  lattice_src_r;
9 layout(binding = 1, rgba32f)  uniform readonly image3D  lattice_src_g;
  layout(binding = 2, rgba32f)  uniform readonly image3D  lattice_src_b;
11
  layout(binding = 3, rgba32f)  uniform writeonly image3D lattice_dst_r;
13 layout(binding = 4, rgba32f)  uniform writeonly image3D lattice_dst_g;
  layout(binding = 5, rgba32f)  uniform writeonly image3D lattice_dst_b;
15
  layout(binding = 6, rgba32f)  uniform readonly image3D  lattice_geometry;
17
  // Propagation values
19 layout(std140, binding = 7) uniform PropagateConfigParams {
```

```
  vec4        transfer_vectors[26];
  vec3        neighbor_directions[26];
  float       outscattering_fractions[26];
};

uniform float       scattering_coeff_normalized_sqrtpi_2;
uniform vec3        propagate_factor_init;
uniform int         use_blocking;

// Constants
const float SQRT_PI_2 = 3.5449077f;

void main(void)
{
  ivec3 cell_dst = ivec3(gl_GlobalInvocationID);
  ivec3 cell_src;

  vec4 cell_r, cell_g, cell_b, cell_bcc;

  vec3 total_outscattered_energy, propagate_factor;
  float blocking_factor;

  vec4 coeffs_r = vec4(0), coeffs_g = vec4(0), coeffs_b = vec4(0);

  for (int d=0; d < 26; d++) {
    cell_src = cell_dst - ivec3(neighbor_directions[d]);

    cell_r = imageLoad(lattice_src_r, cell_src);
    cell_g = imageLoad(lattice_src_g, cell_src);
    cell_b = imageLoad(lattice_src_b, cell_src);
    cell_bcc = imageLoad(lattice_geometry, cell_src);

    total_outscattered_energy = vec3(cell_r.x, cell_g.x, cell_b.x);
    total_outscattered_energy *= scattering_coeff_normalized_sqrtpi_2;
    blocking_factor = clamp(1.f - dot(cell_bcc, transfer_vectors[d]) *
        use_blocking, .0f, 1.f);

    propagate_factor = propagate_factor_init;
    propagate_factor.r *= dot(cell_r, transfer_vectors[d]);
    propagate_factor.g *= dot(cell_g, transfer_vectors[d]);
    propagate_factor.b *= dot(cell_b, transfer_vectors[d]);
    propagate_factor += total_outscattered_energy * outscattering_fractions[d
        ];
    propagate_factor *= blocking_factor;
    propagate_factor = max(propagate_factor, .0f);

    coeffs_r += transfer_vectors[d] * propagate_factor.r;
    coeffs_g += transfer_vectors[d] * propagate_factor.g;
    coeffs_b += transfer_vectors[d] * propagate_factor.b;
  }

  imageStore(lattice_dst_r, cell_dst, coeffs_r);
  imageStore(lattice_dst_g, cell_dst, coeffs_g);
  imageStore(lattice_dst_b, cell_dst, coeffs_b);
}
```

Listing 5.4: OpenGL fragment shader to compute direct illumination in a deferred pass.

```
#version 420

// from pass-quad.vs
in blockVertex {
  vec2 uv;
} inVertex;

out vec4 color;
out float gl_FragDepth;

// Camera GBuffer
uniform sampler2D cam_position_tex;
uniform sampler2D cam_normal_tex;
uniform sampler2D cam_diffuse_tex;
uniform sampler2D cam_depth_tex;

uniform sampler2DShadow light_sm_tex;

// light_direction is assumed to be normalized
uniform vec3 light_position;
uniform vec3 light_direction;
uniform vec4 light_spectrum;
uniform float light_fov;
uniform float light_flux;
uniform mat4 light_vp_matrix;
uniform int light_attenuation;
uniform float light_extinction;

uniform vec3 cam_position;

const float M_PI = 3.14159265358979323846f;
const float M_1_PI = 0.318309886f;        // 1.f / M_PI
const float M_1_PI_8 = 0.039788736f;      // 1.f / (M_PI * 8.f)
const float M_1_PI_4 = 0.079577472f;      // 1.f / (M_PI * 4.f)

const float SM_BIAS = 0.0005f;

/**
 * Compute the normalized Blinn-Phong BRDF, derived by Sloan and Hoffmann 2008
 *
 * See "Real-Time Rendering", 3rd ed, Akenine-Moeller, p.257
 *
 * Note: this function can be optimized
 */
vec3 normalizedBlinnPhongBRDF(vec3 c_diff, vec3 c_spec, float cos_theta_h,
    float m)
{
  c_diff *= M_1_PI;
  c_spec *= (m + 8) * M_1_PI_8;
  return c_diff + c_spec * pow(cos_theta_h, m);
}

/**
 * Spotlight attenuation factor, similar to D3D fixed function impl
 *
 * See "Real-Time Rendering", 3rd ed, Akenine-Moeller, p.221
```

```
 */
float f_dist_spotlight(float cos_s, float cos_u, float cos_p, float s_exp)
{
  float f = 1.f;

  if (cos_s < cos_u) {
    f = 0.0f;

  } else if (cos_u < cos_s && cos_s < cos_p) {
    f *= pow( (cos_s-cos_u)/(cos_p-cos_u), s_exp);
  }

  return f;
}

/**
 * Distance attenuation with Beer's law
 */
float f_dist_beerslaw(float r)
{
  return exp(-light_extinction * r);
}

/**
 * Distance attenuation with squared
 *
 * Params:
 * l = light - surface vector
 */
float f_dist_squared(vec3 l)
{
  return 1.f / dot(l,l);
}

/**
 * Linear distance attenuation
 *
 * Note: not physically based
 */
float f_dist_linear(float r, float r_start, float r_end)
{
  if (r < r_start) {
    return 1.f;

  } else if (r > r_end) {
    return 0.f;

  } else {
    return (r_end - r) / (r_end - r_start);
  }
}

/**
 * Compute shadow mapping attenuation factor
 *
 * Returns 0.f or 1.f
 */
```

```
float f_sm(vec3 pos)
{
  vec4 surfel_cp_light = light_vp_matrix * vec4(pos,1.f);
  vec4 surfel_ndc_light = surfel_cp_light / surfel_cp_light.w;
  vec4 surfel_shadow_coord = surfel_ndc_light * .5f + .5f;
  surfel_shadow_coord.z -= SM_BIAS;
  return texture(light_sm_tex, surfel_shadow_coord.xyz);
}

void main(void)
{
  vec3 p = texture(cam_position_tex, inVertex.uv).xyz;
  vec4 normal_4 = texture(cam_normal_tex, inVertex.uv);
  vec3 n = normal_4.xyz;
  vec3 diffuse = texture(cam_diffuse_tex, inVertex.uv).rgb;
  vec3 specular = vec3(.05f);

  float surface_flux = normal_4.w;
  float light_fov_1_2 = light_fov * .5f;
  float cos_theta_u = cos(radians(light_fov_1_2));
  float intensity = light_flux * M_1_PI_4;
  vec3 lp = light_position - p;
  vec3 l = normalize(lp);
  vec3 v = normalize(cam_position - p);
  vec3 h = normalize(l + v);
  float cos_theta_h = clamp(dot(n, h), 0.f, 1.f);
  float cos_theta_p = cos(radians(light_fov_1_2 - 3.f));
  float cos_theta_s = clamp(dot(light_direction, -l), 0.f, 1.f);
  float cos_theta_i = dot(l, n);

  float dist_atten = 1.f;

  if (light_attenuation == 1) {
    dist_atten = f_dist_beerslaw(length(lp));
  }

  vec3 radiance_brdf = normalizedBlinnPhongBRDF(diffuse, specular, cos_theta_h
      , 50.f);
  vec3 radiance = radiance_brdf;
  radiance *= intensity * f_dist_spotlight(cos_theta_s, cos_theta_u,
      cos_theta_p, 2.f)
            * dist_atten
            * f_sm(p)
            * clamp(cos_theta_i, 0.f, 1.f);
  radiance *= light_spectrum.rgb;
  radiance += (surface_flux * .1f) * radiance_brdf;

  color = vec4(radiance, 1.f);
  gl_FragDepth = texture( cam_depth_tex, inVertex.uv).x;
}
```

CD-ROM Contents

This thesis is accompanied by a CD-ROM containing application sources, configurations, scenes, final thesis presentation and this document.

Application Sources

The presented LPV based application for multi-display installations is archived in `/src/src.tar.gz`. It is built with CMake 2.8+ and it requires two arguments:

- `-DCMAKE_BUILD_TYPE=Release|Debug` The build type of the application.
- `-DLIBRARY_PATH_PREFIX=`⟨`path`⟩ The path to a directory containing required custom libraries in the directories `eq-1.7`, `eq-1.7.debug`, `glew-1.10` and `glew-1.10.debug`.

To build the application, change directory into `src/release` and execute

`cmake .. -DCMAKE_BUILD_TYPE=Release -DLIBRARY_PATH_PREFIX=`⟨`path`⟩.

This script checks for required dependencies. In case dependencies are found, the path is echoed. Otherwise an error is generated. After successful initialization, run `make -j8` where 8 is the number of available CPU processors.

Table 1 lists all custom dependencies. In case the prefix is defined, this library has to be present in ⟨path⟩ in a separate directory with the schema

⟨`Prefix`⟩ − ⟨`Version`⟩⟨`BuildTypeSuffix`⟩,

where ⟨`BuildTypeSuffix`⟩ either equals an empty string for release builds, or it equals `.debug` for debug builds. GCC 4.7+ is tested.

Configurations and Scenes

The file `/src/assets.tar.gz` contains all configurations, GLSL shaders and models required to run the application in a multi-display environment. To run the application in local mode, execute

`./eqLpv -c ../../assets/config/input.ini -s ../../assets/config/scene.ini`.

Library	Version	Prefix
GLEW	1.10	glew
Equalizer	1.7	eq
Tclap	1.2.1	tclap

Table 1: Required dependencies for the application source.

For the CAVE at the LRZ, run `assets/scripts/ini-sles.sh` to initialize environment. Verify that environment variables are properly set. This script starts DTrack2 and unclutter to remove cursors. Next execute `assets/scripts/start-cluster.sh`. This script starts the VRPN server and then the render cluster. See this script for more details. Change the home directory in this script if required. Note that Equalizer creates log files for each render node in the current working directory.

Presentation

The presentation (along with its source files) is located at `/Folien`. Building the presentation from source requires Texlive 2013 or 2014.

Thesis

This document is located at `/Dokumentation/PDF/somm14.pdf`. To build this thesis, run `/Dokumentation/make.sh`. Note that Texlive 2013 or 2014 is required.

List of Figures

List of Tables

Listings

Abbreviations

BRDF	Bi-Directional Reflectance Distribution Function	6
BVH	Bounding Volume Hierarchy	10
CAVE	CAVE Automatic Virtual Environment	1
DAG	Directed Acyclic Graph	35
FOV	Field-of-View	46
G-Buffer	Geometry Buffer	9
GI	Global Illumination	1
HDR	High Dynamic Range	25
HUD	Heads Up Display	45
IBO	Index Buffer Object	40
ILPV	Incremental Light Propagation Volume	vii
IR	Instant Radiosity	9
ISM	Imperfect Shadow Map	10
LDR	Low Dynamic Range	26
LOD	Level-of-Detail	13
LPV	Light Propagation Volume	vii
LRZ	Leibniz Supercomputing Centre	31
MAE	mean absolute error	28
MP	Multiprocessor	36
PCF	Percentage-Closer Filtering	24
PRT	Precomputed Radiance Transfer	8
P-State	Performance State	39
RE	Rendering Equation	6
RSM	Reflective Shadow Map	9
RTE	Radiative Transfer Equation	6
SH	Spherical Harmonics	8
SM	Shadow Map	10
SPD	Spectral Power Distribution	22
STL	C++ Standard Template Library	32
VAL	Virtual Area Light	10
VBO	Vertex Buffer Object	40
VPL	Virtual Point Light	9
VR	Virtual Reality	1

Symbols

σ_a	absorption coefficient	6
$b_{cc'_j}$	blocking factor	13
$\mathcal{B}_{cc'_j}$	blocking distribution function	12
$\mathbf{c}_{\text{diff}}$	diffuse term in BRDF	23
σ_t	extinction coefficient	6
Φ	flux	5
γ_{dsp}	display gamma	26
γ_{enc}	encoding gamma	26
γ_{e2e}	end-to-end gamma	26
$\mathbf{h}$	normalized half-vector	23
λ_{incr}	incremental LPV coefficient	27
I	intensity	14
E	irradiance	5
d_s	lattice cell size	21
$\mathbf{l}$	normalized light vector from surface point to light	23
$\mathbf{n}$	normalized normal	23
γ	outscattering fraction	15
L	radiance	5
$\overline{L}$	normalized radiance	25
σ_s	scattering coefficient	6
SH_{cos}	cosine lobe SH coefficients function	21
Ω	solid angle	5
$\mathbf{c}_{\text{spec}}$	specular term in BRDF	23
Γ	light transfer function	12
v_{cave}	CAVE velocity in world space	45
$\mathbf{v}$	normalized view vector from surface point to eye	23
V	visibility function	24

Bibliography

[AHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008, p. 1045.

[BBH13] T. Barak, J. Bittner, and V. Havran. "Temporally Coherent Adaptive Sampling for Imperfect Shadow Maps". In: *Computer Graphics Forum* 32.4 (2013), pp. 87–96.

[Bim+06] O. Bimber, A Grundhofer, T. Zeidler, D. Danch, and P. Kapakos. "Compensating Indirect Scattering for Immersive and Semi-Immersive Projection Displays". In: *Virtual Reality Conference, 2006*. Mar. 2006, pp. 151–158.

[Bør+11] Jesper Børlum, Brian Bunch Christensen, Thomas Kim Kjeldsen, Peter Trier Mikkelsen, Karsten Østergaard Noe, Jens Rimestad, and Jesper Mosegaard. "SSLPV: Subsurface Light Propagation Volumes". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: ACM, 2011, pp. 7–14.

[Bro+06] Raymond C. Browning, Emily A. Baker, Jessica A. Herron, and Rodger Kram. "Effects of obesity and sex on the energetic cost and preferred speed of walking". In: *Journal of Applied Physiology* 100.2 (2006), pp. 390–398.

[BSA12] Markus Billeter, Erik Sintorn, and Ulf Assarsson. "Real-time Multiple Scattering Using Light Propagation Volumes". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. Costa Mesa, California: ACM, 2012, pp. 119–126.

[Cha60] S. Chandrasekhar. *Radiative Transfer*. Dover Books on Intermediate and Advanced Mathematics. Dover Publications, 1960.

[CLP93] John C Chai, HaeOk S Lee, and Suhas V Patankar. "Ray effect and false scattering in the discrete ordinates method". In: *Numerical Heat Transfer, Part B Fundamentals* 24.4 (1993), pp. 373–389.

[CSD93] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. "Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE". In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: ACM, 1993, pp. 135–142.

[Dac+14] Carsten Dachsbacher, Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Bruce Walter, and Jan Novák. "Scalable Realistic Rendering with Many-Light Methods". In: *Computer Graphics Forum* 33.1 (2014), pp. 88–104.

[DBB06] P. Dutré, K. Bala, and P. Bekaert. *Advanced Global Illumination*. Ak Peters Series. A K Peters, Limited, 2006.

[Dim12] Aleksander Dimitrijević. "Performance State Tracking". In: *OpenGL Insights*. Ed. by Patrick Cozzi and Christophe Riccio. CRC Press, 2012, pp. 527–534.

[Dmi+04] Kirill Dmitriev, Thomas Annen, Grzegorz Krawczyk, Karol Myszkowski, and Hans-Peter Seidel. "A CAVE System for Interactive Modeling of Global Illumination in Car Interior". In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '04. Hong Kong: ACM, 2004, pp. 137–145.

[Don+09] Zhao Dong, Thorsten Grosch, Tobias Ritschel, Jan Kautz, and Hans-Peter Seidel. "Real-time Indirect Illumination with Clustered Visibility." In: *Proc. VMV*. 2009.

[DS05] Carsten Dachsbacher and Marc Stamminger. "Reflective Shadow Maps". In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. Washington, District of Columbia: ACM, 2005, pp. 203–231.

[DS06] Carsten Dachsbacher and Marc Stamminger. "Splatting Indirect Illumination". In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. Redwood City, California: ACM, 2006, pp. 93–100.

[Eis+11] Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. *Real-Time Shadows*. A.K. Peters, 2011, p. 398.

[Eva98] K Franklin Evans. "The spherical harmonics discrete ordinate method for three-dimensional atmospheric radiative transfer". In: *Journal of the Atmospheric Sciences* 55.3 (1998), pp. 429–446.

[Fat09] Raanan Fattal. "Participating Media Illumination Using Light Propagation Maps". In: *ACM Trans. Graph.* 28.1 (Feb. 2009), 7:1–7:11.

[Gei+04] Robert Geist, Karl Rasche, James Westall, and Robert Schalkoff. "Lattice-Boltzmann Lighting". In: *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*. EGSR'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 355–362.

[Gre+06] Paul Green, Jan Kautz, Wojciech Matusik, and Frédo Durand. "View-dependent Precomputed Light Transport Using Nonlinear Gaussian Function Approximations". In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. Redwood City, California: ACM, 2006, pp. 7–14.

[GS10] Iliyan Georgiev and Philipp Slusallek. "Simple and Robust Iterative Importance Sampling of Virtual Point Lights". In: *Eurographics 2010-Short Papers*. The Eurographics Association. 2010, pp. 57–60.

[Hil12] Sébastien Hillaire. "Improving Performance by Reducing Calls to the Driver". In: *OpenGL Insights*. Ed. by Patrick Cozzi and Christophe Riccio. CRC Press, 2012, pp. 353–364.

[Hoa+10] Roger Hoang, Steve Koepnick, Joseph D. Mahsman, Matthew Sgambati, Cody J. White, and Daniel S. Coming. "Exploring Global Illumination for Virtual Reality". In: *ACM SIGGRAPH 2010 Posters*. SIGGRAPH '10. Los Angeles, California: ACM, 2010, 125:1–125:1.

[Hol+11] Matthias Hollander, Tobias Ritschel, Elmar Eisemann, and Tamy Boubekeur. "ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination". In: *Computer Graphics Forum* 30.4 (2011), pp. 1233–1240.

[Jen96] HenrikWann Jensen. "Global Illumination using Photon Maps". English. In: *Rendering Techniques 1996*. Ed. by Xavier Pueyo and Peter Schroeder. Eurographics. Springer Vienna, 1996, pp. 21–30.

[Kaj86] James T. Kajiya. “The Rendering Equation”. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 143–150.

[KD10] Anton Kaplanyan and Carsten Dachsbacher. “Cascaded Light Propagation Volumes for Real-time Indirect Illumination”. In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’10. Washington, D.C.: ACM, 2010, pp. 99–107.

[KED11] Anton Kaplanyan, Wolfgang Engel, and Carsten Dachsbacher. “Diffuse Global Illumination with Temporally Coherent Light Propagation Volumes”. In: *GPU Pro 2*. Ed. by Wolfgang Engel. A. K. Peters, Ltd., 2011, pp. 185–203.

[Kel96] Alexander Keller. “Quasi-Monte Carlo Radiosity”. English. In: *Rendering Techniques ’96*. Ed. by Xavier Pueyo and Peter Schröder. Eurographics. Springer Vienna, 1996, pp. 101–110.

[Kel97] Alexander Keller. “Instant Radiosity”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 49–56.

[Kie53] Jack Kiefer. “Sequential minimax search for a maximum”. In: *Proceedings of the American Mathematical Society* 4.3 (1953), pp. 502–506.

[Lai+07] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. “Incremental Instant Radiosity for Real-time Indirect Illumination”. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR’07. Grenoble, France: Eurographics Association, 2007, pp. 277–286.

[Löf+11] Alexander Löffler, Lukas Marsalek, Hilko Hoffmann, and Philipp Slusallek. “Realistic Lighting Simulation for Interactive VR Applications”. In: *Virtual Environments 2011 - Proceedings of the Joint Virtual Reality Conference of euroVR and EGVE (JVRC)*. Nottingham, UK: Eurographics Association, Sept. 2011, pp. 1–8.

[Loo+12] Bradford J. Loos, Derek Nowrouzezahrai, Wojciech Jarosz, and Peter-Pike Sloan. “Delta Radiance Transfer”. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’12. Costa Mesa, California: ACM, 2012, pp. 191–196.

[ML09] Morgan McGuire and David Luebke. “Hardware-accelerated Global Illumination by Image Space Photon Mapping”. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG ’09. New Orleans, Louisiana: ACM, 2009, pp. 77–89.

[MM02] Vincent C. H. Ma and Michael D. McCool. “Low Latency Photon Mapping Using Block Hashing”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS ’02. Saarbrucken, Germany: Eurographics Association, 2002, pp. 89–99.

[NED11] Jan Novak, Thomas Engelhardt, and Carsten Dachsbacher. “Screen-space Bias Compensation for Interactive High-quality Global Illumination with Virtual Point Lights”. In: *Symposium on Interactive 3D Graphics and Games*. I3D ’11. San Francisco, California: ACM, 2011, pp. 119–124.

[NRH03] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. "All-frequency Shadows Using Non-linear Wavelet Lighting Approximation". In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 376–381.

[NW09] Greg Nichols and Chris Wyman. "Multiresolution Splatting for Indirect Illumination". In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. Boston, Massachusetts: ACM, 2009, pp. 83–90.

[OD13] John David Olovsson and Michael Doggett. "Octree Light Propagation Volumes". In: *SIGRAD 2013* (2013), p. 27.

[Pan+07] Minghao Pan, Rui Wang Xinguo Liu, Qunsheng Peng, and Hujun Bao. "Precomputed Radiance Transfer Field for Rendering Interreflections in Dynamic Scenes". In: *Computer Graphics Forum* 26.3 (2007), pp. 485–493.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[Poy03] Charles Poynton. *Digital Video and HDTV Algorithms and Interfaces*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[Pur+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. "Photon Mapping on Programmable Graphics Hardware". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '03. San Diego, California: Eurographics Association, 2003, pp. 41–50.

[Rit+08] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. "Imperfect Shadow Maps for Efficient Computation of Indirect Illumination". In: *ACM Trans. Graph.* 27.5 (Dec. 2008), 129:1–129:8.

[Rit+12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. "The State of the Art in Interactive Global Illumination". In: *Computer Graphics Forum* 31.1 (2012), pp. 160–188.

[Seg+06] B. Segovia, J. C. Iehl, R. Mitanchey, and B. Péroche. "Bidirectional Instant Radiosity". In: *Proceedings of the 17th Eurographics Conference on Rendering Techniques*. EGSR'06. Nicosia, Cyprus: Eurographics Association, 2006, pp. 389–397.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. "Precomputed Radiance Transfer for Real-time Rendering in Dynamic, Low-frequency Lighting Environments". In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 527–536.

[TS06] Yu-Ting Tsai and Zen-Chung Shih. "All-frequency Precomputed Radiance Transfer Using Spherical Radial Basis Functions and Clustered Tensor Approximation". In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 967–976.

[Yao+10] Chunhui Yao, Bin Wang, Bin Chan, Junhai Yong, and Jean-Claude Paul. "Multi-Image Based Photon Tracing for Interactive Global Illumination of Dynamic Scenes". In: *Computer Graphics Forum* 29.4 (2010), pp. 1315–1324.

[Zho+08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. "Real-time KD-tree Construction on Graphics Hardware". In: *ACM Trans. Graph.* 27.5 (Dec. 2008), 126:1–126:11.