NTNU – Trondheim
Norwegian University of
Science and Technology

Progressive Photon Mapping on GPUs

Stian Aaraas Pedersen

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

PROJECT DESCRIPTION

Progressive Photon Mapping on GPUs

Ray tracing is a method capable of producing much more realistic images than rasterization. Global illumination methods consider both illumination directly from light sources, as well as reflections from other surfaces in the scene. However, the computational costs associated with exploring all paths of light are huge; it can take hours to render high quality images of complex scenes.

Taking advantage of the computational power of modern GPUs, NVIDIA OPTIX, a generic GPU ray tracing engine, runs on top of the NVIDIA CUDA parallel computing architecture. This project will investigate how well global illumination calculations using a recent technique known as *memoryless* progressive photon mapping can be off-loaded onto GPUs. This will be done by implementing the method in the Nvidia OPTIX ray-tracing framework, and comparing its strengths and weaknesses to the original progressive photon mapping algorithm. Demonstrating effects such as depth-of-field and/or motion blur, and discussion of pros and cons will be included. In addition, volumetric (participating media) effects like smoke and fog will be considered, with the aim of implementing it in the NTNU HPC-lab Snow Simulator. If time permits, the thesis will pursue creating a CUDA implementation of the algorithm and compare it with to OPTIX with regards to speed, especially on multi-GPU systems.

ABSTRACT

Physically based rendering using ray tracing is capable of producing realistic images of much higher quality than other methods. However, the computational costs associated with exploring all paths of light are huge; it can take hours to render high quality images of complex scenes. Using graphics processing units has emerged as a popular way to speed up this process. The recent appearance of libraries like Nvidia's CUDA and OptiX make the processing power of modern GPUs more available than ever before.

This project includes an overview of current photon mapping techniques. We present a complete render application based on photon mapping which runs entirely on the GPU. Several different photon map implementations suitable for modern GPU architectures are considered and evaluated. A uniform grid approach based on photon sorting on the GPU is found to be fast to construct and efficient for photon gathering.

The application is extended to support *volumetric effects* like fog and smoke. Our implementation is tested on a set of benchmark scenes exhibiting phenomenon like global illumination, reflections and refractions, and participating volumetric media.

A major contribution of this thesis is to demonstrate how recent advances in photon mapping can be used to render an image using many GPUs simultaneously. Our results show that we are able to get close to linear speedup employing up to six GPUs in a distributed system. We can emit up to 18 million photons per second on six Nvidia GTX 480 and generate images of our test scenes with little to no noise in a few minutes. Our implementation is straightforward to extend to a cluster of GPUs.

SAMMENDRAG

Fysisk-basert rendering ved bruk av stråle-simulering kan produsere realistiske bilder av en mye høyere kvalitet enn andre metoder. Dessverre er beregningskostnadene knyttet til utforskning av lysbaner store: det kan ta mange timer å generere bilder av kompliserte scener og effekter. Bruk av skjermkort (GPU) for å akselerere denne prosessen har blitt svært populært de siste årene. Lanseringen av bibliotek og verktøy som Nvidias CUDA og OptiX gjør prosesseringskraften til moderne skjermkort tilgjengelig for flere.

Dette prosjektet inneholder en gjennomgang av aktuelle teknikker basert på "photon mapping". Vi presenterer en komplett applikasjon som kjører i sin helhet på skjermkortet. Flere alternativer for datastrukturer i "photon mapping" er testet og vurdert. Å oppdele scenen i et rutenett og så sortere fotonene viser seg å være raskt å konsturere, og effektivt å bruke.

Vår applikasjon er så utvidet for å støtte *volumetriske effekter* som røyk og tåke. Vi tester vår implementasjon på et sett av testscener som demonstrerer global illuminasjon, refleksjoner og refraksjoner, og volumetriske media.

Et stort bidrag fra denne oppgaven er å demonstrere hvordan nylige utvidelser av photon mapping-metoden kan brukes til å generere et bilde med bruk av flere skjermkort samtidig. Våre resultater viser at vi får nær lineær speedup ved bruk av opp til seks GPU-er i et distribuert system. Vi kan sende ut 18 millioner fotoner per sekund fra lyskildene med seks Nvidia GTX 480 samtidig. Bilder av våre testscener med lite støy genereres i løpet av få minutter. Vår implementasjon kan enkelt utvides for å støtte en klynge av skjermkort.

ACKNOWLEDGMENTS

I would like to thank Dr. Anne Cathrine Elster for supervising me for the duration of this Master's thesis, for invaluable feedback on this report, and also for her efforts as a manager of the NTNU HPC-lab. The HPC-lab certainly would not exist if it wasn't for her. Special thanks go out to all the supporters of the HPC-lab, especially Nvidia which on several occasions has donated graphics cards and resources through their CUDA Research Center and CUDA Teaching Center programs. The work carried out in this project would not been possible without their support.

Last but not least, thanks goes out to all the co-students at the HPC-lab who have made the duration of this project much more enjoyable.

Stian Pedersen,
June 13, 2013

CONTENTS

1

INTRODUCTION

Efficient simulation of *global illumination* is a standing research problem in computer graphics. It involves exploring all kinds of light transport in a scene to solve the *rendering equation* [Kaj86]. Informally, the light upon any point depends on the light reflected or emitted at every other point in the entire universe. Many of the most popular global illumination algorithms are based on *Monte Carlo* methods, which use random sampling to evaluate a function at sample points to estimate the true solution.

Photon mapping [JC98] is one method able to efficiently capture effects due to specular reflections and refractions, which often are difficult to find with other, unbiased methods based on path tracing. Light due to *specular-diffuse-specular* paths are particularly problematic [HOJ08]. Photon mapping caches a number of Monte Carlo photon samples emitted from the light sources, and reuses these samples in a second step to estimate indirect illumination at any point. This estimation introduces bias in the form of a blurring of the image, which is reduced with the number of photons used. Progressive photon mapping [HOJ08] is a variant of photon mapping which removes the memory constraint by executing the algorithm in iterations.

While the computational power of processors continuously increases at impressive rates, the demands of the audience grow just as fast. A paradox known as *Blinn's Law*[1] states that render times remain constant despite advances in hardware and algorithms. Pixar's Toy Story (1995), the first fully computer-animated feature film, required on average 7 processor-hours to render - per frame [Bet12]. For Cars 2 (2011) from the same studio, some of the most demanding frames involving ray-tracing took as much as 80 to 90 processor-hours to render [Ter11]. Motion-blur, depth-of-field, high-definition resolution and 3D, as well as realistic lighting, reflections, hair, fur, skin, cloth, fog, smoke, fire and so on are expected of modern pictures. These phenomenon are in no way cheap to model realistically. The sheer computational workload required to render complicated scenes to high quality may take days, even on high-end computers. Clearly, performance is a major concern: 3D artists and visual architects would prefer to have interactive editing of objects, materials and lights in a scene. Slow feedback hinders the creative process and prevent experimentation. Even a tiny reduction in render times can accumulate to thousands of dollars saved for a big production.

1 Jim Blinn is known for his work as a computer graphics expert at NASA's Jet Propulsion Laboratory.

Modern GPUs are manycore architectures with thousands of cores which can be exploited to get impressive performance at a budget. These cards are first and foremost designed for the gaming industry, but a trend the last years has been to use these cards to accelerate other algorithms of a parallel nature. The GPU has become more widely available due to the appearance of APIs like CUDA and OpenCL.

Ray tracing is well suited for the GPU since it is inherently a slow operation and each pixel can be rendered in parallel. The GPU has attracted the interest of rendering scientists and professionals, and a number of new rendering algorithms well suited for their architecture [HOJ08, KZ11] have been developed. In this thesis, we investigate how the rendering process can be offloaded efficiently onto GPUs.

1.1 CONTRIBUTION

Our focus is on implementing the *photon mapping* method [Jen09, HOJ08] entirely on the GPU. The Nvidia OptiX ray-tracing engine is chosen as a foundation for our renderer. We present a complete application available for others to test and experiment with. Different photon map structures, a CPU-based *k*-d tree, a uniform grid based on sorting, and a stochastic hash are considered. We analyze their construction times as well as photon gathering performance. Volumetric effects like fog and smoke are described and implemented in the renderer. Our suite of test scenes is benchmarked on three different GPUs, including the recent Nvidia Tesla K20. Finally, we present a distributed renderer for the first time able to execute photon mapping using many GPUs simultaneously with impressive efficiency. Results are presented for up to 6 Nvidia GeForce GTX 480 rendering a single image in parallel.

1.2 REPORT OUTLINE

The rest of the report is structured as follows:

- Chapter 2 presents related work.
- Chapter 3 contains background material on ray-tracing and physically based rendering. Main focus is on the (progressive) photon mapping algorithm and rendering of volumetric effects.
- Our implementation is described in detail in Chapter 4.
- We present results and rendered images on a set of benchmark scenes in Chapter 5.
- Chapter 6 contains conclusive remarks.
- Finally, Chapter 7 mentions interesting future work.

2

RELATED WORK

This section presents earlier work related to ray tracing and photon mapping on the Graphics Processing Unit.

2.1 RAY TRACING ON GPU

Numerous articles have been written on ray tracing on the Graphics Processing Unit. Some of the earliest work done was in 2002 with the "Ray Engine" [CHH02], implementing only the ray-triangle intersection on the GPU.

In 2002, Purcell *et al.*[PBMH02] investigated ray tracing on the programmable graphics hardware that was emerging at the time. Previously, graphics hardware were fixed function pipelines and difficult to program. Now, customizable vertex and fragment programs meant you could use the GPU for other things than rasterization. Purcell *et al.* implemented a four pass ray tracing algorithm: ray generation, traversing, intersection and shading. However, they were still limited by the fact that the programmable framework was mapped to graphics problems. Their application was written as a set of fragment programs, using textures for storage and the stencil buffer to control conditional execution. Due to the difficulties of implementing better acceleration structures, such as *k*-d trees or bounding volume hierarchies, they used a simple uniform grid. Still, they demonstrated that it was possible to move the entire ray tracing algorithm over to the GPU.

Foley and Sugerman [FS05] presented the first *k*-d tree traversal algorithm that could be efficiently run on the GPU. Their approach was able to significantly outperform uniform grids. Since the GPU could not efficiently support stacks, several stack-less traversal algorithms was suggested.

Horn et al. [HSHH07] were the first to present interactive frame-rates using the GPU. They based their implementation on a rasterizer with a ray tracer for secondary and shadow rays. Since GPU thread-local memory was scarce, a *k*-d tree short-stack approach with a stack-less fallback was their method of choice.

In 2007, Nvidia launched the Compute Unified Device Architecture (CUDA) parallel computing architecture [Nvi07]. Since its release, CUDA has been widely utilized for parallel ray tracing research. In this period, the GPU took major steps away from a fixed-function rasterization pipeline to a programmable and dynamic unit applicable to many problems.

Nvidia OptiX was unveiled at SIGGRAPH 2009 [Nvi09]. OptiX is a framework built on CUDA with the purpose of simplifying development of ray-tracing

based applications. Holger Ludvigsen [LE10] explored OptiX the same year. He found it to be flexible and capable of rendering scenes at interactive frame rates. Ludvigsen also discovered that it gives near perfect speed-up on several GPUs in certain scenarios. However, at the time, the framework was found to be 3-5 times slower than comparable, hand-optimized ray tracers on similar scenes and hardware.

2.2 PHOTON MAPPING ON GPU

Purcell *et al.* described the first implementation of photon mapping for GPU in 2003 [PDC+03]. They used a grid-based photon map where the photons are sorted into their correct cell. Since sorting at the time was slow to perform on the GPU, a *stencil routing* approach which directs photons to their final destination was suggested.

Hachisuka *et al.* [HOJ08] presented the progressive photon mapping method in 2008, which removed the memory constraint. Since GPUs usually have less available memory than the CPU, this approach is more suitable for GPUs.

Several implementations of photon mapping has appeared the latest years, especially after the release of CUDA [Nvi07]. Fleisz [Fle09] implemented photon mapping on the GPU using CUDA in 2009 and considered several alternatives of photon maps.

In 2009, McGuire and Luebke [ML09] presented an approach based on *image-space* photon mapping combining the CPU and GPU. This approach exploits the fact that the initial camera and light rays/photons have a common center of projection. This assumes point lights and a pinhole camera, but is able to get interactive rates at high resolutions.

Knaus and Zwicker [KZ11] introduced a "memoryless" progressive photon mapping approach in 2011, which effectively removed the dependency between iterations in PPM.

Alongside the OptiX framework Nvidia also released a small demo using progressive photon mapping [Nvi09].

Hachisuka and Wann Jensen [HJ10] described an GPU implementation where the photons are stochastically stored in a hash table, which avoids any kind of list generation at the cost of increased variance.

Jarosz *et al.* introduced photon beams [JNSJ11] to improve rendering of participating media. They also described a *progressive* version of photon beams [JNT+11], drawing insight from memoryless PPM. Several implementations was mentioned, and one of these implementations was based on the OptiX framework.

Mara *et al.* [MLM13] consider screen-space photon mapping techniques for interactive applications. Interactive frame rates is achieved for some scenes and some conditions, at the cost of amortizing photon tracing among several frames and applying reconstruction filters to remove noise.

Distributed rendering (using several computers communicating over a network) is desirable to speed up the rendering process. Animated sequences can be rendered each distinct frame in parallel. We consider rendering of a *single* image on multiple CPUs and/or GPUs at the same time, since this requires a lower level of work division.

V-Ray from Chaos Software [Cha13], Indigo Renderer from Glare Technologies [Gla13], and Luxrender [Lux13] are some of the available renderers which support network acceleration. These can also accelerate (a subset of) the rendering process to the GPU. In March 2013, Otoy announced a GPU cloud-based version of their Octane Render [Oto13]. A demo was presented at the GTC 2013 conference where 112 GPUs were utilized at the same time. All these renderers are unbiased, i.e. based on *path-tracing* and derived algorithms, and not photon mapping.

Knaus and Zwicker [KZ11] rendered an image on a heterogeneous cluster of 166 nodes using progressive photon mapping. The individual images was rendered using `pbrt` [PH04] and then averaged. A speedup of 4.5 was achieved over a single CPU. To the best of our knowledge, there is no photon mapping-based renderer which can utilize several GPUs simultaneously.

3

BACKGROUND

This chapter contains material to back up our work. To keep this chapter succinct, some of the material is just briefly touched upon for readers with no background in ray tracing or rendering. Some of the more advanced topics and analytical proofs are outside the scope of this thesis, but references are included so the reader can look it up.

The rest of this chapter is structured as follows;

- Section 3.1 introduces some probability theory we base ourselves on for the rest of this chapter.
- Section 3.2 touches upon the concept of ray-tracing, the ray equation and acceleration structures.
- Section 3.3 describes radiometry, a set of quantities and equations that govern light transport.
- Section 3.4 deals with light-surface interactions and some common material types.
- Section 3.5 introduces global illumination, including the rendering equation and the path tracing algorithm.
- Section 3.6 goes on to describe photon mapping, with focus on recent advantages in progressive photon mapping.
- Section 3.7 discusses participating media.
- Section 3.8 contains a brief introduction to the Graphics Processing Unit (GPU). CUDA, Nvidia's GPU programming framework, is also mentioned.
- Section 3.9 describes OptiX, a framework for ray-tracing built on top of CUDA, which we employ in this thesis.

This section briefly introduces some of the probability concepts that we'll use later to describe rendering algorithms.

3.1.1 *Monte Carlo methods*

Monte Carlo methods (named after the Monte Carlo casino in Monaco) rely on repeated random sampling in order to find an approximate solution to a problem.

The Monte Carlo estimator to compute an integral $I = \int f(x)\,dx$ is [DBB06]

$$\hat{I} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}$$

where $p(x)$ is the probability distribution function. From this, it is easy to see that $\mathrm{E}(\hat{I}) = \int f(x)\,dx = I$. The Monte Carlo method is independent of the dimensionality of the integrand [PH04]; however, it has a slow convergence of $\mathcal{O}(\sqrt{N})$. Monte Carlo is applicable for problems where a closed-form solution is impractical or even impossible. This is often the case with rendering algorithms. In fact, the path tracing algorithm introduced later in this chapter is an integral of *infinite dimensionality.*

3.1.2 *Bias and consistency of estimators*

We briefly touch the concepts of estimator bias and consistency. We'll denote a fixed parameter as X, and our estimator for X as $\hat{X}_n$, where n is the number of sampled used for calculating the estimate of X. The bias of the estimator $\hat{X}_n$ is $E(\hat{X}_n) - X = E(\hat{X}_n - X)$, which can be considered the expected value of the *error* of the estimate. The estimator is *unbiased* if the expected error is zero; $E(\hat{X}_n - X) = 0$. Otherwise, the estimator is *biased*. A biased estimator $\hat{X}_n$ is *consistent* if $\lim_{n\to\infty}\hat{X}_n = X$, that is, if it converges to the correct value X.

Unbiased estimators have the advantage that the expected error is zero after any number of samples. Increasing the number of samples will typically reduce the variance of the estimate. A biased but consistent estimator's error vanishes only in the limit.

3.1.3 *Russian roulette*

The Russian roulette technique [PH04] can increase the probability that an evaluated sample will have a significant contribution to the result. Intuitively, it is wasted work to calculate many samples that have close to no impact on the end result. However, we cannot simply ignore these calculations without introducing bias.

Russian roulette is applied by introducing a termination probability q. Before the evaluation of an expensive function, we stop with a probability q. With prob-

ability $1 - q$, the evaluation is still performed, but scaled by a factor $\frac{1}{1-q}$. The expected value of the modified estimator remains the same as the original. Since Russian roulette methods increase variance, the choice of q is very important. $1 - q$ should be proportional to the importance of the evaluation to the final result.

3.2 RAY TRACING

Ray tracing is an image-generation technique where rays are generated from a virtual camera and into a scene. A renderer will simulate how these rays interact with objects in the scene to produce an image. Usually, ray tracing is recursive: each surface hit can spawn a number of new rays, on reflection off specular surfaces, or refraction through transmissive materials. Methods based on screen-space projection (*rasterization*) have problems accurately capturing reflections and shadows. Since ray tracing model how light propagate (at a simplified level), it captures these effects by nature.

The first ray tracing algorithm was introduced by Appel [App68] in 1968. His idea was to shoot rays from the eye, through each pixel of an image plane, and find the closest object intersecting the path of the ray. Appel would stop his algorithm when the ray hit the closest object, which we today call "ray casting". In 1979, Whitted [Whi80] took the next step, by making it a recursive process.

3.2.1 *The ray equation*

A ray is defined by its origin o and direction d, parameterized by a distance parameter t, $R(t) = o + td$. Using a *pinhole* camera, primary rays are generated so that they start at the eye and pass through the pixels of some defined image plane. More complicated camera models are able to capture focus *(depth-of-field*).

3.2.2 *Geometry and acceleration structures*

Since ray tracing simulates how rays interact with geometry, the most basic operation is ray-geometry intersection tests. Objects usually have some mathematical description allowing them to be intersected against rays directly. Some common types of objects are spheres, cylinders, quadrangles and cubes. The most important primitive, however, is the triangle, as larger models usually are built up from triangles. The most common and effective ray-triangle intersection algorithms are based on *barycentric coordinates* [Wal06].

A naive linear-time intersection algorithm would test a ray against every object in the scene. Since large scenes may contain millions of objects, this approach quickly becomes infeasible. Acceleration structures are space subdivision hierarchies that substantially reduce intersection time. Their idea is to group objects that are close in larger entities, and skip bunches of objects at a time if they are nowhere near the ray. Common acceleration structures include k-d trees and bounding volume hierarchies.

Symbol	Description
x, y	Positions
ψ, ω	Directions, $d\omega$ often serving as the differential solid angle
θ, ϕ	Planar angles
t	A distance along a ray
$\Omega_{4\pi}$, $\Omega_{2\pi}$	The sphere and hemisphere of directions
$L(x \rightarrow \psi)$	Exitant radiance from x in outgoing direction ψ
$L(x \leftarrow \psi)$	Incident radiance at x in incoming direction ψ
$f_r(x, \omega \leftrightarrow \psi)$	The surface BRDF at x between directions ω and ψ
$p(x, \omega \leftrightarrow \psi)$	The phase function at x between directions ω and ψ
ζ	A random number uniformly distributed between 0 and 1
σ_s, σ_a, σ_t	Scattering, absorption and extinction coefficients
$T_r(x \leftrightarrow y)$	Transmittance between points x and y

Table 1: The notation we use throughout this thesis.

3.3 RADIOMETRY

The goal of rendering algorithms is to produce photo-realistic images that accurately capture the appearance of objects in a scene. Radiometry is the field of physics that addresses the measurement of light propagation and reflection. Radiometry describes light at the *geometrical optics* level, where it travels as rays. Effects due to the wave nature of light, such as polarization, diffraction and interference, are not captured by radiometry without extending the framework. However, dealing with everyday objects that are substantially larger than the wavelength of light this approximation is good.

Photometry, a distinct but related field of science, covers measurement of light as it is *perceived* by the human visual system. The visual response of the human eye to the frequency range of visible light has been standardized, and photometry takes this standard into account. Since photometric quantities can be derived from their corresponding radiometric units, rendering systems prefer to work with radiometry units internally. The process of converting into photometric units is usually delayed to the last stage of the render pipeline.

Text-books on physically based rendering will typically cover radiometry thoroughly. We base this chapter mainly on [DBB06] and [PH04]. The reader is encouraged to consult these works for more information.

3.3.1 *Radiant Flux*

The basic unit in radiometry is radiant flux, often called radiant power, denoted Φ. Radiant flux is radiant energy per unit time and is expressed in watts (joules/second). Radiant flux captures how much energy flows through a surface per time.

Emission from light sources is usually declared using radiant flux. For instance, a light source can emit 50 watts of flux, while a table can receive 200 watts of flux.

3.3.2 *Irradiance and Radiant Exitance*

Irradiance, noted E, represent the incident power on a surface per unit surface area. The unit is watts/m^2. For example, if 100 watts of power is incident on a table with an area of $2m^2$, the irradiance on each surface point is $50\,\text{watts}/m^2$.

A similar term is radiant exitance M, used when the power is leaving as opposed to arriving at a surface. An area light source of $0.2m^2$ which emits 100 watts has a radiant exitance of $500\,\text{watts}/m^2$.

3.3.3 *Radiance*

Radiance L (Fig. 3.3.1) is defined as radiant flux per unit projected area per unit solid angle, watts/(sr·m^2)

$$L = \frac{\mathrm{d}^2\Phi}{dA\, d\omega \cos\theta} \tag{3.3.1}$$

Notice that the cone $d\omega$ will hit the surface point at an angle. The cross-section of the cone and the surface is a differential area dA. The area dA will be "smeared out" over a larger surface area when the ray comes from a grazing angle. The $\cos\theta$ term takes this increased area into account.

Radiance is the most important quantity in rendering algorithms because it captures how surfaces *appear* from the viewpoint of the observer. In vacuum, radiance remains constant as it propagates along rays, so it is a natural unit to compute in ray tracing. Looking at a red car from close up and far away, it will appear equally bright. However, far away the car subtends a smaller solid angle, so the power that reaches the eye is less. Finally, all the other quantities can be derived in terms of integrals of radiance over directions and surfaces.

We use a notation which is based on [DBB06], listed in Table 1. Table 2 contains the basic physical units as a reference for the next sections.

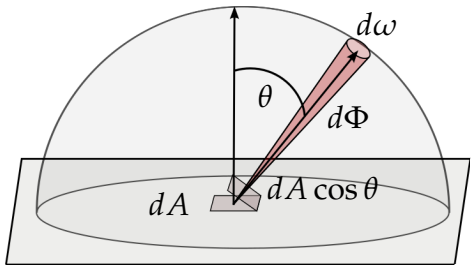

Figure 3.3.1: Radiance.

Name	Symbol	Unit	Description
Radiant energy	Q	Joule	Energy
Radiant power	Φ	Watt	Energy per unit time
Irradiance	E	Watt/m^2	Power incident on a surface
Radiant exitance	M	Watt/m^2	Power emitted from a surface
Radiance	L	Watt/(sr $\cdot$ m^2)	Power per unit solid angle per unit projected area

Table 2: Units in radiometry.

3.4 LIGHT-SURFACE INTERACTION

Some materials, like metals or glossy plastics, have a mirror-like appearance. "Matte" materials like painted walls appear about the same from any angle. This section will describe how we model these light-surface interactions.

A function which defines the distribution of light reflection at a surface is called a reflectance distribution function. In the most general case, light which enters a surface at a point p in direction ψ may leave at another point q in a new direction ω. The function $f_r(p, \psi, q, \omega)$ defining the ratio of incident and exitant radiance between (p, ψ) and (q, ω) is called the *bidirectional surface scattering reflectance distribution function* (BSSRDF).

We can simplify the function under the assumption that the light arriving at some point leaves the surface at the same point, i.e. we ignore *subsurface scattering*. The simplified function is called the bidirectional reflectance distribution function (BRDF), and is defined as the ratio between the differential radiance in exitant direction ω, and the differential irradiance incident through a differential solid angle $d\psi$ [DBB06];

$$f_r(p, \psi \to \omega) = \frac{dL(x \to \omega)}{dE(x \leftarrow \psi)} = \frac{dL(x \to \omega)}{L(x \leftarrow \psi)(N_x \cdot \psi)\, d\psi}$$

The BRDF is a four-dimensional function defined at each point on the surface, two dimensions (azimuth and zenith angles) for each of the directions.

Physically based BRDFs have some additional properties. The Helmholtz Reciprocity property implies that the value of the BRDF is unchanged if we exchange incident and exitant directions. $f_r(p, \psi \to \omega) = f_r(p, \omega \to \psi)$. Therefore, we use the notation $f_r(p, \psi \leftrightarrow \omega)$.

Conservation of energy dictates that a surface cannot reflect more energy than it receives,

$$\forall x, \psi \int_\Omega f_r(x, \omega \leftrightarrow \psi) \cos\theta d\omega \leq 1$$

It is essential that these properties hold true for physically-based materials, or else the assumptions of the rendering algorithm may break down.

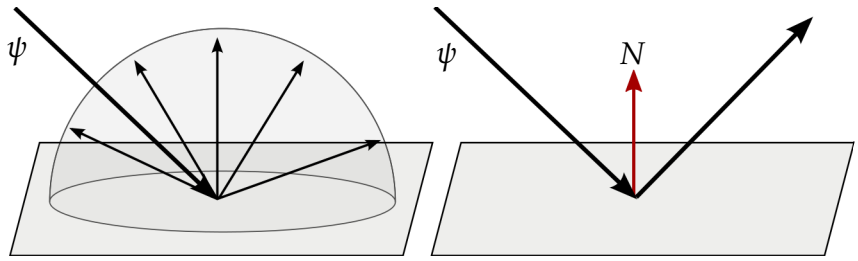

Figure 3.4.1: Perfectly diffuse (left) and perfectly specular (right) surfaces.

3.4.1 *Diffuse surfaces*

The simplest material type is an ideal diffuse BRDF (Fig. 3.4.1), often called *Lambertian* [DBB06]. A diffuse material reflects light uniformly across the hemisphere, independently of the incoming direction. To an observer, a diffuse material looks the same from every viewing direction. No real-world material is a perfect diffuse reflector; however, it is a good approximation for many day-to-day materials like matte walls and ceilings. The BRDF for a diffuse material is

$$f_r(p, \psi \leftrightarrow \omega) = \frac{\rho_d}{\pi}$$

where ρ_d is the *reflectivity* of the material, a number between 0 and 1 that represents the fraction of the energy which is reflected.

3.4.2 *Specular surfaces*

A perfectly specular surface will reflect or refract light in one direction only (Fig. 3.4.1). A mirror is an example of a specular surface. The law of reflection gives the direction of the reflected ray from incident direction ψ and the normal of the surface N:

$$R = 2(N \cdot \psi)N - \psi$$

Glass materials can both reflect and refract light. The direction of refraction can be found using Snell's law, while the ratio between reflected and refracted light is given by Fresnel's equations [DBB06, PH04].

3.4.3 *Shading models*

Real-life materials are neither perfectly diffuse nor perfectly specular, but rather something in between. A number of shading models have been described in literature [DBB06, PH04]. Examples of popular models are Phong, Blinn-Phong and Cook-Torrance. They differ both in their sophistication and cost of implementation. Some materials, like metals, react with light in intricate ways that are difficult to describe accurately. Materials like plastics and wallpaper can be approximated with simpler BRDFs.

This section focuses on methods of finding *global illumination*. We can consider illumination as composed of two parts;

- Direct illumination includes the light that comes directly from a light source. Direct illumination is cheap to calculate, by finding the light sources that illuminate a point in the scene and accumulate their individual radiance contributions.
- Indirect illumination includes the light that has reflected off other surfaces at least once. Indirect illumination is more expensive to calculate, but adds to the realism of a scene. In computer games, indirect illumination is typically approximated using an ambient term. The ambient term can be considered as the average indirect illumination on a surface, and is usually estimated by level designers.

Global illumination is the sum of the contributions from direct and indirect illumination. Global illumination methods are tasked with finding the total illumination in a scene.

3.5.1 *The rendering equation*

"The rendering equation" was first coined by Kajiya in 1986 [Kaj86]. The rendering equation generalized an existing number of algorithms into a simple integral equation, and has since been considered as a fundamental equation in physically based rendering.

Some assumptions are made by the rendering equation:

- The rendering equation assumes a *steady-state system* of *radiance equilibrium*. The law of conservation of energy states that the total energy in such a system remains constant.
- We let light travel instantly, a reasonable assumption at the scale we are operating. This implies that the steady state is achieved instantly.
- We assume the absence of *participating media*. Light travel between surfaces along straight lines in *vacuum*.
- Finally, we ignore *sub-surface scattering*, where radiance entering a surface at position p may scatter below the surface and leave at another location q.

At each surface point x and direction ψ, the rendering equation formulates exitant radiance $L(x \to \psi)$. The rendering equation is given as

$$\begin{aligned} L(x \to \psi) &= L_e(x \to \psi) + L_r(x \to \psi) \\ &= L_e(x \to \psi) + \int_{\Omega_{2\pi}} f_r(x, \psi \leftrightarrow \omega) L(x \leftarrow \omega) \cos\theta \, d\omega \end{aligned} \tag{3.5.1}$$

where L_e is the emitted radiance and L_r is the reflected radiance from x in outgoing direction ψ. The integral in the second line is over all incoming hemispherical directions ω at x. f_r is the BRDF of the surface, and θ is the angle between ω and the surface normal N_x, $\cos\theta = N_x \cdot \omega$.

Notice how the rendering equation makes no separation by radiance directly from light sources, and radiance reflected from other surfaces, they are both captured by L. This unification makes the rendering equation so concise. $L_e(x \to \psi)$ is non-zero for light sources (*emitters*), while most materials only reflect incoming radiance as specified by the BRDF.

At this point we should note that the radiance is dependent on wavelength λ, which can be considered an extra parameter to the radiance and BRDF functions. Wavelength is usually implicit in these equations. In rendering systems, we typically sample the rendering equation to obtain a RGB color triplet.

3.5.2 *Using radiance for ray-tracing*

If we want to render a physically based image with full global illumination, we need to find the total radiance incident on each pixel of the image plane. This can be described by the following integral over the pixel:

$$L_{pixel} = \int_{imageplane} L(p \to \psi)h(p)dp = \int_{imageplane} L(r(p \to \psi) \to -\psi)h(p)dp$$

where p is a point on the image plane, $h(p)$ is a filter/weight function, and $r(p \to \psi)$ is the ray-trace function which returns the point visible from point p looking in direction ψ. Usually, $h(p)$ is a box filter so that the final radiance value for a pixel is the average incident radiance over the image plane pixel footprint. $h(p)$ can also contain the area of a lens to capture depth-of-field.

3.5.3 *Path tracing*

Photon mapping is the main topic of this thesis, and will be covered in detail in the next section. First we briefly introduce path tracing because it is simple to explain, unbiased, and the foundation for more complex methods like bidirectional path tracing [LW93] and Metropolis light transport [VG97]. Some of the same techniques used in path tracing are also used for photon mapping, including direct sampling of light sources in the scene.

Path tracing estimates the integral for

$$L_r(x \to \psi) = \int_{\Omega} f_r(x, \psi \leftrightarrow \omega)L(x \leftarrow \omega)\cos\theta\, d\omega$$

in the rendering equation (3.5.1) using Monte Carlo integration (Sec. 3.1.1). The idea is simply to pick random directions ω over the hemisphere $\Omega_{2\pi}$, distributed according to a probability density function $p(\omega)$.

The estimator for L_r is [DBB06]

$$\begin{aligned}\langle L_r(x \to \psi)\rangle &= \frac{1}{N}\sum_{i=1}^{N}\frac{f_r(x,\psi \leftrightarrow \omega_i)L(x \leftarrow \omega_i)(\omega_i \cdot N_x)}{p(\omega_i)} \\ &= \frac{1}{N}\sum_{i=1}^{N}\frac{f_r(x,\psi \leftrightarrow \omega_i)L(r(x,-\omega_i) \to \omega_i)(\omega_i \cdot N_x)}{p(\omega_i)} \quad (3.5.2)\end{aligned}$$

$r(x,\omega)$ is the closest surface point from x in direction ω, and is found by ray tracing. Now we have to perform another radiance estimation of $L(r(x,-\omega_i) \to \omega_i)$ using the rendering equation recursively. These recursive estimations form a path traced through the scene. This path may be infinite, so we need a stopping criterion. We use Russian roulette (3.1.3) to find an unbiased solution of an infinite series using a finite number of evaluations. The idea is to sample paths based on their importance to the final scene. Therefore, we pick the termination probability q based on surface properties. A dark surface will absorb more energy compared to a bright surface. Therefore, the probability of termination on a dark surface should be higher.

The simplest choice of the probability density function $p(\omega)$ is the uniform $p(\omega) = 1/(2\pi)$. However, it is inefficient to sample many directions near the horizon where $\cos\theta = (\omega \cdot N_x) \approx 0$. Therefore, we use a cosine-distribution $p(\omega) = \cos\theta/\pi$. This choice of $p(\omega)$ prevents us from having to calculate $\cos\theta$ explicitly, and, in the case of a diffuse BRDF, it will cancel out the $1/\pi$ factor.

We can split L_r into L_{direct} and $L_{indirect}$ and sample light sources directly. We exploit that we already know the location of the light sources in the scene. This technique reduces noise, since a path does not have to end with an emitter to give radiance contribution. To evaluate the emitted radiance from a light source upon a point x, we must transform the integral over hemisphere to an integral over surface points y on the light. This gives the following integral for incident radiance [DBB06]

$$\begin{aligned}L_{direct}(x \to \psi) &= \int_A f_r(x,\psi \leftrightarrow \vec{yx})L_e(y \to \vec{yx})V(x,y)G(x,y)dA_y \\ G(x,y) &= \cos(N_x,-\vec{yx})\cos(N_y,\vec{yx})/r_{xy}^2\end{aligned}$$

where $\vec{yx}$ is the direction from y to x, and V is a visibility term, 1 if x and y are mutually visible and 0 otherwise.

The efficiency of path tracing is connected to the probability of hitting an emitter. If emitters are small and encapsulated in a specular fixture (like a tiny light bulb in a large stadium), the probability that a path starting from the camera will end up at the light is microscopic. This is an important problem with naive path tracing. It is still unbiased, so after a sufficient amount of time the correct image will appear. Derived algorithms like bidirectional path tracing [LW93] and Metropolis light transport [VG97] tries to connect light and camera paths smarter to speed up this process.

Figure 3.5.1: Top image: full global illumination demonstrated on the Conference Room test scene. Middle image: only light *directly* from the light sources contributes. Bottom image: Only indirect radiance (light that has bounded at least one time, multiplied with the surface BRDF).

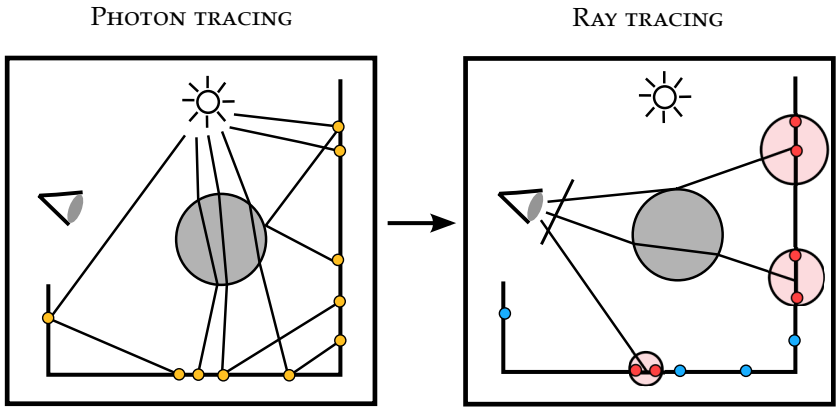

Figure 3.6.1: Photon mapping.

3.6 PHOTON MAPPING

Photon mapping is a global illumination method developed by Wann Jensen [Jen96, Jen09]. It is considered robust to many difficult lighting scenarios which is almost impossible to capture with unbiased methods (based on path tracing). Illumination due to *specular-diffuse-specular* paths are particularly problematic [HOJ08]. Photon mapping excels for *caustics* formed by focus of glass materials. It is in fact the only practical method to find many light paths common in everyday scenes, for instance, the combination of a glass light bulb and a glossy floor seen through a lens camera. Photon mapping captures these effects by combining light rays and camera rays via a kernel.

The remainder of this sub-chapter will go on to describe the photon mapping method in detail.

3.6.1 *The original PM algorithm*

Photon mapping is a two-phased algorithm;

In the first pass, flux packets named photons are emitted out from the light sources in the scene. They react with surfaces the same way as rays, they can be reflected or transmitted, and some of the energy in the photon may be absorbed, depending on the properties of the material. When photons react with surfaces, they are deposited in a data structure called the *photon map*. The data that is stored with each photon includes hit position x, flux/power $\Delta\Phi$, and incoming photon direction ω_i.

In the second pass, ray tracing is carried out. Rays are traced from the eye onto surfaces in the scene. Typically, direct illumination is calculated directly tracing shadow rays towards all light sources. If the point in question is visible from a light, the incoming radiance contribution from that light is accumulated. Indirect illumination, on the other hand, is estimated using the photons stored in the photon map. The photon map needs to support fast retrieval of the N closest photons to any query location. Therefore, the photon map is often implemented as a k-d tree.

Radiance estimation

To estimate the radiance at a point x on a surface, we locate the closest N photons in the photon map and use them for the radiance estimate. The idea behind radiance estimation in PM is to expand a sphere around x until it contains N photons and use these photons to estimate the value of the radiance at x. $\Delta\Phi_p$ is the flux carried by photon p, ω_p is the incoming direction of the photon, and r is the radius of the sphere which contains all N photons. Combining the rendering equation (3.5.1) and the definition of radiance (3.3.1), we get

$$\begin{aligned} L(x \to \psi) &= \int_\Omega f_r(x, \psi, \omega) L(x \leftarrow \omega) \cos\theta \, d\omega \\ &= \int_\Omega f_r(x, \psi \leftrightarrow \omega) \frac{d^2\Phi}{dA d\omega \cos\theta} \cos\theta \, d\omega \\ &\approx \sum_{p=1}^{N} f_r(x, \psi \leftrightarrow \omega_p) \frac{\Delta\Phi_p}{\pi r^2} \end{aligned} \tag{3.6.1}$$

Under the assumption that the surface is locally flat, dA is in the last step approximated as the projected surface area of the sphere, πr^2.

BIAS IN PHOTON MAPPING The radiance estimation step works as a low-pass filter and introduces bias to the image. It connects light and camera paths via a non-zero bandwidth kernel. The error is manifested as low-frequency noise (blur) in the rendered image. The bias is unfortunate, since unbiased solutions are considered the gold standard. However, photon mapping is consistent, so the error will vanish if an infinite number of photons are used. Both photon mapping and unbiased methods require an infinite number of samples to reach the solution. For a small number of samples, low-frequency blur is more pleasing to the eye than high-frequency variance (salt and pepper noise).

3.6.2 *Progressive photon mapping*

In photon mapping, each and every photon used must be in memory for the duration of the algorithm. Therefore, photon mapping is both time and memory constrained. For complex scenes, hundreds of millions to billions of photons may be required to reach acceptable quality.

Progressive photon mapping was introduced by Hachisuka *et al.* in 2008 [HOJ08], and then extended to support stochastic effects[1] in 2009 [HJ09]. For completeness and discussion, we'll first describe the original PPM algorithm (including the stochastic extension), then consider newer improvements.

Progressive photon mapping removes the memory constraint by performing the radiance estimation in *iterations*. Each iteration use a new, randomly traced

1 Stochastic/distributed ray tracing is a refinement of ray tracing that allows us to model "soft" phenomena. Stochastic effects include anti-aliasing, depth-of-field, motion-blur, and glossy reflections.

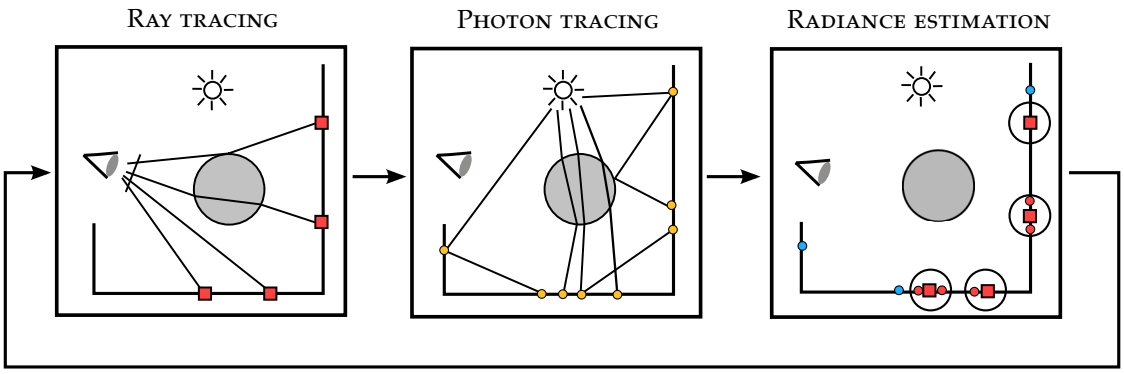

Figure 3.6.2: Progressive photon mapping.

set of photons to update the radiance estimate in the scene. The photons are tossed after the end of an iteration. The bounded memory requirement makes the algorithm well suited for GPUs, which are efficient compute units with fast (but limited) memory.

The idea behind progressive photon mapping is to reorder the original photon mapping by *first* performing ray-tracing, *then* photon tracing, and finally radiance estimation. This sequence is repeated in iterations. The rendered image is *progressively* more accurate.

RAY TRACING PASS Find points in the scene visible through each pixel of the camera. Each pixel footprint is denoted as a *region*, and each ray tracing pass finds a new surface point inside this region. If we encounter a surface which spawns two rays, like a glass material that both reflects and refracts, we use probability sampling to keep the path linear. Each region stores hit position p, the direction of the incoming ray ω, surface BRDF identifier, and surface normal n. Every region (pixel) includes space for three values used in the algorithm: current photon radius R, a count of photons N, and accumulated unnormalized power τ. These values are passed on from one iteration to the next.

PHOTON TRACING Photons are emitted from the light sources in the scene and stored in the photon map. Photons may be absorbed, reflected or transmitted when they hit surfaces. In fact, this procedure is performed identically to regular photon mapping. A constant number of photons are emitted each pass, which makes the memory requirement bounded.

ESTIMATION PASS After the ray tracing pass, we have a set of new surface points in the scene visible through the camera. A new set of photons was generated in the photon tracing step. We loop through hit points and accumulate power from photons that are within a distance R_i. We use these new photons to refine our radiance estimate for the pixel.

These passes are then repeated any number of times. After an estimation pass it is possible to render an approximate image. As more iterations are executed, the

quality of the radiance estimate is improved. The image will converge towards the correct solution. This property is useful since we can stop the algorithm at a point where the image is of acceptable quality.

Algorithm

We use the recently traced photons to update the radiance estimate for a pixel. Each pixel stores current photon radius R, a current number of photons N, and accumulated power τ. Initially $N_0 = 0$, $\tau_0 = 0$, and R_0 is set to some scene-dependent initial radius.

After photon tracing iteration i, and for each scene hit point x, we find all photons that are within a distance R_i of x. Assume we have N_i photons accumulated prior to this iteration, and we have just found M new photons to add to N_i. Note that M can be zero if there are no photons within the radius. In the limit, we need an infinite number of photons within an infinitely small radius to have the correct solution. Our goal is to reduce the radius R while simultaneously increase N, the number of photons that are within this radius.

We let the updated value of N_i be N_{i+1} and find it as,

$$N_{i+1} = N_i + \alpha M \tag{3.6.2}$$

α is a parameter which controls what fraction of the M photons we should include in our estimate. If the radiance estimate of photon mapping is to converge to the correct solution, we must have an increase in the number of photons. Therefore, $\alpha > 0$. Also, we don't want to add more than M photons. Therefore, we let $\alpha = (0, 1)$. Recently, Kaplanyan and Dachsbacher [KD13] found the asymptotically optimal α for shrinking the radius to be $\alpha = 2/3$, although the choice of R_0 is still important for the convergence rate.

We also need to reduce R_i. Under the assumption that the photon density is *uniform* within the radius, [HOJo8] shows that we can find the value R_{i+1} as

$$R_{i+1} = R_i \sqrt{\frac{N_i + \alpha M}{N_i + M}} \tag{3.6.3}$$

Since the square root factor must be less than or equal to one, R cannot increase. At this point, we have found the new values R_{i+1} and N_{i+1} and have simultaneously ensured that N increases or stays the same while R decreases or stays the same between iterations. In case we have found any photons within R, i.e. $M > 0$, the radius will always be reduced.

Radiant flux estimate

The estimate τ_M for the total unnormalized flux from the M new photons is given as

$$\tau_M = \sum_{p=1}^{M} f_r(x, \psi \leftrightarrow \omega_p) \Delta\Phi'_p \tag{3.6.4}$$

where $\Delta\Phi'_p$ is the unnormalized power carried by the photon p. Unnormalized in this sense implies that we don't divide by the number of emitted photons at this step, which is an important difference from regular photon mapping.

At any iteration i, we combine τ_i and τ_M to find τ_{i+1}. We need to take into account that the radius R_i decreases between iterations. By assuming that the power density is uniform inside the disc, the reduction of power is equal to the reduction in area. This gives

$$\tau_{i+1} = (\tau_i + \tau_M)\frac{R_{i+1}^2}{R_i^2} \tag{3.6.5}$$

Radiance estimate

The previous sections have described how we update the statistics needed for the progressive photon mapping algorithm. This section will explain how these statistics are used to estimate radiance for any iteration i. The radiance estimate is

$$\begin{aligned}
\hat{L}(x \to \psi) &= \int_\Omega f_r(x, \psi \leftrightarrow \omega) L(x \leftarrow \omega) \cos\theta \, d\omega \\
&\approx \frac{1}{\Delta A} \sum_{p=1}^{N} f_r(x, \psi \leftrightarrow \omega_p) \Delta\Phi_p \\
&\approx \frac{\tau_i}{\pi R_i^2} \frac{1}{N_{emitted,\,i}}
\end{aligned} \tag{3.6.6}$$

The radiance $L(x \to \psi)$ is the limit as i goes towards infinity:

$$L(x \to \psi) = \lim_{i \to \infty} \frac{\tau_i}{\pi R_i^2} \frac{1}{N_{emitted,\,i}}$$

The radiance estimate is the same as in regular photon mapping (Eq. 3.6.1), except that we have moved the division by $N_{emitted,\,i}$ to the last step. This is necessary since $N_{emitted,\,i}$ increases each iteration and we need to ensure that photons emitted in all iterations are equally important.

The values R_i and τ_i are stored in the per-pixel data structure. $N_{emitted,\,i}$ is the total number of photons emitted from the light sources after i iterations of the algorithm, so $N_{emitted,\,i} = i \cdot N_{\text{emitted\_per\_iteration}}$.

Memoryless Progressive Photon Mapping

Unfortunately, the PPM algorithm is dependent on the maintenance of local statistics (R_i and N_i) per pixel. These values need to be passed over from an iteration to the next, preventing any possible parallelization across iterations. A recent reformulation of progressive photon mapping by Knaus and Zwicker [KZ11] proves that PPM can be executed without keeping these statistics.

The authors took a "probabilistic approach", studying the error ϵ_i of each iteration of the algorithm.

By letting the variance of ϵ_i increase by a factor

$$\frac{\mathrm{Var}(\epsilon_{i+1})}{\mathrm{Var}(\epsilon_i)} = \frac{i+1}{i+\alpha}$$

the authors show that the variance and the expected value of the average error, $\mathrm{Var}(\bar{\epsilon}_i)$ and $\mathrm{E}(\bar{\epsilon}_i)$, both converge to 0.

By proving and using the fact that the variance between to iterations is *inversely proportional* to the square radius used in the estimate,

$$\frac{r_{i+1}^2}{r_i^2} = \frac{\mathrm{Var}(\epsilon_i)}{\mathrm{Var}(\epsilon_{i+1})} = \frac{i+\alpha}{i+1} \tag{3.6.7}$$

we can find the sequence of per-iteration radii in the algorithm. [KZ11] contains a proof which shows that the new radii reduction scheme is equivalent to that of the original PPM [HOJ08]. Since this new sequence is independent of the number of photons we find in any iteration, the authors have in fact proved that the rate of radii reduction is independent of local statistics. Therefore, gathering the statistics is no longer necessary, which simplifies implementation and reduces the memory requirement. Since there is no data to carry along from one iteration to the next, these iterations can be run completely in parallel.

The novel memoryless PPM approach was compared with the previous (stochastic) version in terms of image quality. Visual results were found to be practically identical and any difference was due to noise. This new fixed radii reduction scheme provides numerous advantages, which we employ in our own work, and no clear disadvantages.

Parallel Progressive Photon Mapping

Knaus and Zwicker [KZ11] pointed to the connection between progressive photon mapping and regular photon mapping. Progressive photon mapping is simply an iterative execution of regular photon mapping with a constant number of photons per iteration. The sequence of radii R_i used in the photon map radiance estimate is predetermined, given by Eq. 3.6.7. We do not have to execute these iterations in any particular order, however, to estimate an iteration i we need every computation up until that point.

The authors briefly describe an implementation that uses multiple CPUs to render an image by *averaging* the individual iterations, using a photon mapper as a "black box". The radiance estimate in progressive photon mapping differs from regular photon mapping only by *normalizing* the result in the last step. This normalization accounts for the total number of photons emitted after any iteration. It is obvious that this is equivalent to taking the average of all radiance estimates.

The rendering equation (Sec. 3.5.1) makes the assumption that light travels in a *vacuum* between surfaces in a scene, so radiance is *conserved* along its path. While this simplification is acceptable for some environments, it rarely holds up in real life. Fog, clouds, smoke and dust are examples of *participating media* which react in different ways with light that traverses through. Imagine driving on the freeway an early, foggy morning, looking at the head lights of the oncoming cars. Light is scattered by water drops and changes directions, which produce interesting visual effects. Even the atmosphere reacts as a participating media, which explains why it appears blue. If we want to render scenes with participating media, we need to extend our mathematical framework to include this more complicated scenario.

In a vacuum, the radiance between two mutually visible points x and y is conserved; $L(x \leftarrow \psi) = L(y \rightarrow \psi)$. A participating medium can affect the radiance from x to y in four different ways [DBB06]:

Volume Emission

Some mediums, like a lighter fire, contribute with additional light to a scene. Energy in the form of a flammable fluid is converted into visible light. The volume emittance function $\epsilon(x)$ [Watt/m^3] describes how much energy is emitted per unit volume per second. Modeling volumetric light sources can create many interesting scenes, for instance, a bon-fire at night, but is outside the scope of this thesis.

Absorption and out-scattering

Photons which travel through a medium may be *absorbed* by particles in the medium. The energy of the photons are then converted into other forms of energy, for example, kinetic energy of the particles. Thick black smoke is an example of a medium which absorbs most of the photons that pass through. The absorption coefficient $\sigma_a(z)$ has units [1/m] and gives the probability of absorption per unit distance travelled in the medium.

The photons that travel along a ray may also collide with particles in the medium and change directions, a phenomenon called *out-scattering*. The scattering coefficient σ_s is similar to the absorption coefficient σ_a, with unit [1/m]. It gives the probability of scattering per unit of distance travelled in the medium. Since both absorption and out-scattering reduce the radiance along a ray, we define the extinction coefficient $\sigma_t = \sigma_a + \sigma_s$. If the extinction coefficient is *uniform* across the medium, the medium is *homogeneous*. If the extinction coefficient varies, the medium is *heterogeneous.*

A photon which travels a small distance ds will have a probability $\sigma_t\, ds$ of being absorbed or out-scattered. The *scatter albedo* $\alpha = \sigma_s/\sigma_t$ gives the probability that an event is an out-scatter rather than an absorption.

We define the *transmittance function* $\tau(x \leftrightarrow y)$ as the fraction of the radiance that remains after traveling from x to y. The transmittance function is a value

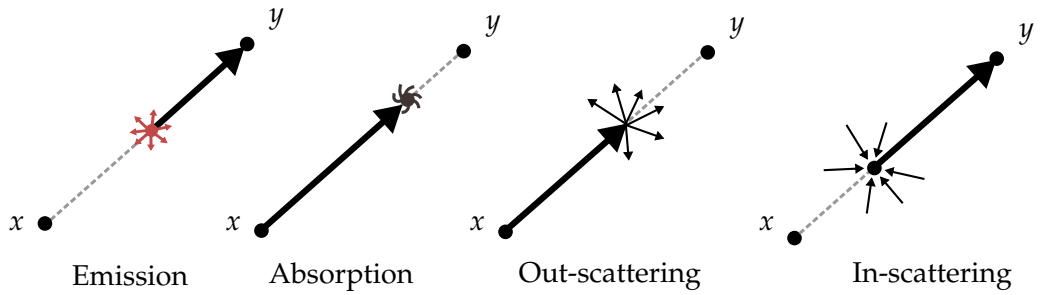

Figure 3.7.1: Radiance in a participating medium may be affected by emission, absorption, out- and in-scattering.

between 0 and 1, where 0 indicates that no radiance is left or, in other words, that every photon has been absorbed or out-scattered. By solving for the differential equation for radiance [DBB06], the transmittance function is

$$T_r(x \leftrightarrow y) = e^{-\int_0^s \sigma_t(x+t\omega)\,dt}$$

where s is the distance in space between the two points, and ω is the direction between them. For homogeneous media, the extinction coefficient σ_t is constant, so the transmittance can be evaluated directly;

$$T_r(x \leftrightarrow y) = e^{-\sigma_t s}$$

The reduced radiance due to absorption and out-scattering in the medium can then be found as

$$L(x \leftarrow \psi) = \tau(x \leftrightarrow y)L(y \rightarrow \psi)$$

When an out-scattering event happens in a medium, the new direction of the photon is given by the *phase function* $p(x, \omega, \psi)$. The phase function defines the probability that a photon which enters in a direction ω will leave in a direction ψ. Since the phase function is a probability distribution, it must be normalized and integrate to 1 across the sphere,

$$\forall x,\ \psi \int_{\Omega_{4\pi}} p(x, \omega, \psi) d\omega = 1$$

If the photon has the same chance of being scattered in every direction, we say that the medium is *isotropic*. In an isotropic medium, the phase function can be found to be the constant $1/4\pi$ [DBB06]. An isotropic medium is the volume equivalent to a diffuse surface. If the phase function is dependent on ω and ψ, the medium is *anisotropic*.

Anisotropic phase functions can be used to model more complicated medias [Jar08]. The *Henyey-Greenstein* phase function is commonly used to model light scattering in clouds [HG41]. It has a single parameter g which controls if the medium is mainly forwards- or backwards scattering. The *Schlick* phase function [BSS93] is often used as an approximation to Henyey-Greenstein since it is cheaper to evaluate. Smaller particles and electrons may scatter light according to the *Rayleigh* phase function [Ray71]. *Lorenz-Mie* [Lor90, Mie08] model particles of size comparable to the wavelength of light.

In-scattering

Just as some photons scatter and change direction along a ray, out-scattered photons from other rays will enter the current one. The in-scattering is described by a volume density $L^{vi}(x \to \psi)$ with unit [Watt/m^{3}sr]. In a small distance ds, the amount of in-scattered radiance is $dL^i(x \to \psi) = L^{vi}(x \to \psi)\, ds$ [DBB06].

The *volume scattering equation* is

$$L^{vi}(x \to \psi) = \sigma_s(x) \int_{\Omega_{4\pi}} p(x, \omega, \psi) L(x \to \omega) d\omega$$

Finding the in-scattered radiance is the most difficult step, since we need to take out-scattered radiance from other rays into consideration.

3.7.1 *The Radiative Transfer Equation*

We have now considered emission, absorption, out-scattering and in-scattering, and we are ready to consider how these combined alter the radiance in a scene. The change in radiance at a point x is [JC98]

$$\frac{dL(x \to \psi)}{dx} = \sigma_a(x) L_e(x \to \psi) + \sigma_s(x) L_i(x \to \psi) - \sigma_t(x) L(x \to \psi)$$

where $L_i(x_t \to \psi)$, the in-scattered radiance at x_t in direction ψ, depends on incoming radiance from the sphere of directions $\Omega_{4\pi}$

$$L_i(x_t \to \psi) = \int_{\Omega_{4\pi}} p(x_t, \omega, \psi) L(x_t \leftarrow \omega)\, d\omega$$

Radiance transfer in participating media is then described by the *radiative transfer equation* [Cha60, JNSJ11]. The radiance for a point x in an incoming direction ψ is

$$\begin{aligned} L(x \leftarrow \psi) &= \tau(x \leftrightarrow y) L(y \to \psi) \\ &\quad + \int_0^s T_r(x \leftrightarrow x_t) \sigma_s(x_t) L_i(x_t \to \psi)\, dt \end{aligned} \tag{3.7.1}$$

In these equations, y is the closest visible surface point, and $L(y \to \psi)$ is the surface radiance at this point (given by the rendering equation), p is the phase function, and s is the distance from x to y, i.e. the distance travelled in the medium.

SUMMARY When we want to solve for the radiance in a participating medium, we consider it to be made up of two terms. The first term is the radiance leaving the closest visible surface in our direction, multiplied by the transmittance τ. The transmittance factor covers the fact that some of the photons will be absorbed or scattered along its flight. The second term and the most difficult to calculate is in-scattered radiance. In-scattering can happen at every location on the path between points x and y. Therefore, the RTE (3.7.1) integrates along the path and accumulates in-scattered radiance at each point, multiplied by the scattering

coefficient σ_s and the transmittance from x to the current point on the path ($x_t = x + (-\omega)t$).

3.7.2 *Volumetric Photon Mapping*

The photon mapping method was extended by Jensen and Christensen [JC98] to consider participating media. Volumetric photon mapping stores photons in a photon map when they scatter in a medium, analogous to photon mapping for surfaces. Since photon mapping is a surface-based algorithm, some adjustments to the radiance estimate are required to consider radiance in a 3D volume. The main difference is that while surface photon mapping uses the projected 2D area in the estimate (Eq. 3.6.1), volumetric photon mapping uses the full volume [JC98].

$$L_i(x \to \psi) \approx \frac{1}{\sigma_s(x)} \sum_{i=1}^{N} p(x, \omega_i, \psi) \frac{\Delta\Phi_i}{\frac{4}{3}\pi r^3} \tag{3.7.2}$$

where $\Delta\Phi_i$ is the power/flux of the photon, and r is the radius of the sphere that contains these N photons. Typically, N is set to some limit. We then perform a nearest-neighbor query to find the closest N photons and estimate the power density. Since the direction of the photon ω_i is stored with each photon, it is also possible to handle anisotropic phase functions.

One way to estimate the volumetric radiance of a ray is to use *ray marching*. Ray marching estimates the integral by small steps Δs and iteratively adjusts the radiance. In-scattered and emitted radiance for the current step is computed, and the radiance from the previous step is reduced due to extinction (out-scattering and/or absorption). At each step, we must query the photon map to find the N nearest photons and calculate the in-scattered radiance with (3.7.2).

3.7.3 *The Beam Radiance Estimate*

Ray marching is costly, especially on GPU architectures. An improvement was introduced by Jarosz *et. al* [JZJo8, JNSJ11], known as the beam radiance estimate.

The beam radiance estimate can estimate the radiance for a ray in one single operation and costly marching is avoided. A ray is extended with a radius to become a *beam* (Fig. 3.7.2). We can consider all volumetric photons which intersect this beam directly. An equivalent dual interpretation is that all volumetric photons are converted into spheres, and each ray is intersected with these spheres. If the ray intersects the sphere, photon contribution is added to the radiance estimate. The sphere is interpreted as a *photon disc* perpendicular to the direction of the ray. A two dimensional kernel is then applied to estimate the density, similar to photon mapping on surfaces. The radius of each beam (or equivalently, the radius of each sphere) controls the noise to blur ratio.

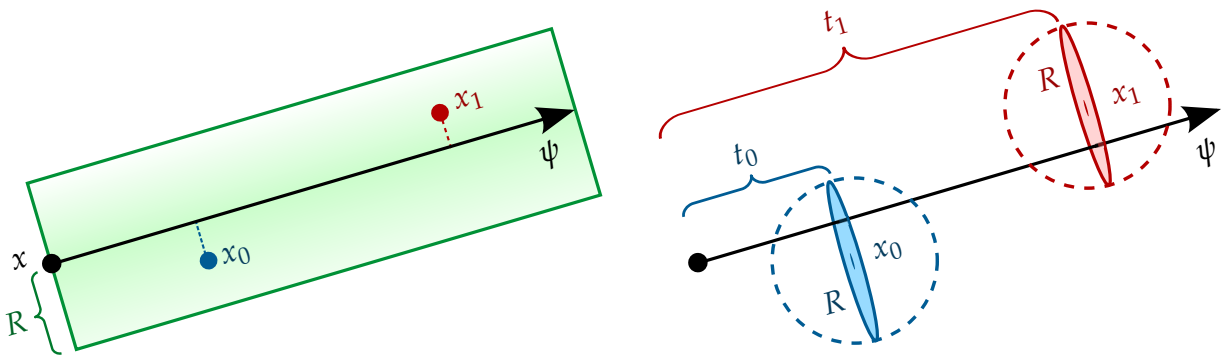

Figure 3.7.2: A *beam* (left) and equivalent disc/sphere interpretation (right).

The beam radiance estimate is [JNSJ11]:

$$L(x \leftarrow \psi) = \frac{1}{K(r^2)} \sum_{i \in B} p(x_i, \omega_i, \psi) \Phi_i e^{-\sigma_t t_i} \tag{3.7.3}$$

where the sum is over all photons which intersect the beam B (direction ψ with a radius r), p is the phase function, Φ_i is the power of photon i, and σ_t is the extinction coefficient of the medium. t_i is the projected distance from x to the photon *along the ray* and can be found using vector projection onto ψ as $t_i = (x - x_i) \cdot \psi$. Finally, K is a kernel to convert into a density, and for cylindrical beams, we let $K(r^2) = \pi r^2$. The beam radiance estimate is an improvement over ray marching since we can enumerate every photon sphere which intersects the ray one time. This is a standard ray-tracing operation, so we can base ourselves on existing accelerated code to perform intersection tests. Beams also have significant quality benefits over marching since every photon is counted. Ray marching may miss photons if the step size is too large. Finally, noise is reduced since the blur dimensionality is 2D rather than 3D.

The success of the GPU, short for *Graphics Processing Unit*, is first and foremost launched by the fast-growing games industry, which has an unending appetite for more power. The GPU is a specialized device used for processing and rasterizing of huge amounts of vertices and triangles at real-time frame rates. It is a many-core device that specializes on high execution throughput of parallel applications.

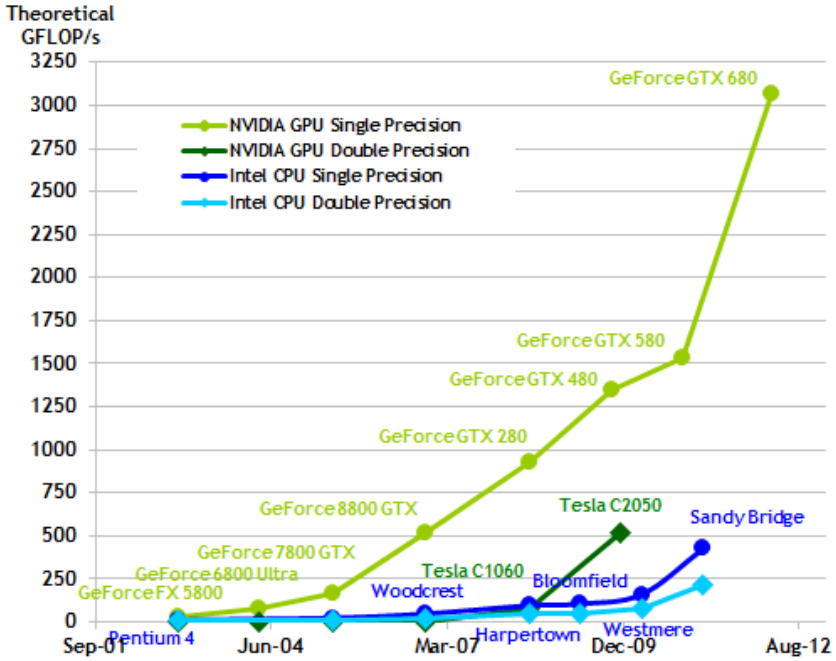

Figure 3.8.1: Theoretical GFLOP/s for Nvidia GPU and Intel CPU architectures shows that the GPU performance has outgrown CPU performance. Graph with permission from Nvidia [NVI13c].

The GPU surpassed the CPU in theoretical maximum FLOP/second performance around 2002-2003 (Fig. **??**). The performance of the GPU has increased much faster than the CPU since - so the gap has grown tremendously. It is expected that the GPU will get even more powerful in the future. Nvidia claim that their Maxwell architecture, slated for a release in 2014, will have a speedup in performance per watt between 2 and 3 [NVI11] over the current Kepler.

The GPU has also surpassed the CPU in number of transistors on the chip. For instance, the GK104 has 3.5 billion, while an Intel i7 Ivy Bridge has around 1.7 billion. The biggest difference between the GPU and the CPU is how the space on the chip is divided. The CPU has advanced branch prediction features, instruction level parallelism, and a sophisticated multi-leveled cache hierarchy to overcome the memory wall. These features take a lot of space on the die. Typical CPUs will consist of 4-8 cores with identical functionality and a multi-level cache. GPUs are simpler in their structure and are built up with more repeated logic. A

GPU mainly consists of an on-chip global RAM memory and an array of identical streaming multiprocessors, each able to switch between hundreds to thousands of threads in almost no time. Most of its transistors are devoted to data processing. The GPUs performance stems from the streaming multiprocessors ability to quickly launch and switch between thousands of threads.

3.8.1 *Compute Unified Device Architecture (CUDA)*

Using the massive power of the GPU is an emerging trend in high-performance computing. The GPU is first and foremost a numeric computing engine, so it cannot be applied for all tasks. The best candidates for GPU optimization inhibits natural "data parallelism", where the same operations are done on many (hundreds of thousands to millions) points of data. Preferably, there should be many arithmetic operations per memory "load" or "store". A number of areas within math, finance, medicine, biology and visualization offer problems that exhibit just this kind of highly parallel nature. These problems can be accelerated to get a solution in faster time - or the increased efficiency could be used to find a more accurate/high-resolution solution, for instance, in medical imaging.

In 2007 Nvidia released CUDA to support joint CPU/GPU applications. This was an important step towards general-purpose programming on the GPU. CUDA is a programming environment that consists of the `NVCC` compiler, libraries, IDE plug ins and useful tools like a debugger and visual profiler. Introductory resources for CUDA information are the C Programming Guide [NVI13c] and Best Practices Guide [NVI13b] from Nvidia. The reader is also suggested to consult [Far11] which contain a complete walkthrough of the functionality offered by CUDA.

3.9 OptiX

OptiX was introduced by Nvidia in 2010 as a "general purpose GPU-based ray tracing engine" [PBD+10]. OptiX is not a full-featured renderer, but a framework which builds upon the observation that most ray-tracing algorithms follow a similar outline and can be implemented as a user-programmable pipeline of operations. OptiX focuses on the low-level operations of ray-tracing and avoids embracing rendering-specific constructs. This philosophy makes OptiX "general purpose", capable of solving many problems which follows the same outline. The creators suggest that OptiX can be used for both interactive and offline rendering algorithms, as well as other problems like collision detection, artificial intelligence and scientific simulations. Since OptiX is built on CUDA, it is available for all systems which have a CUDA-enabled GPU.

OptiX provides many useful features while abstracting away the low-level details of ray-tracing on a GPU. Therefore, we use it for our implementation in this thesis.

3.9.1 *The OptiX pipeline*

The OptiX programmable pipeline resembles graphics libraries like OpenGL and DirectX. The user-programmable units in OptiX are simply called *programs*. There are several program types which are executed at different stages of the pipeline:

RAY GENERATION programs are the entry point into the pipeline. Their purpose is to produce rays that are to be traced in the scene. In a rendering application, the ray generation program will generate the camera ray using, for instance, a pinhole model. Each ray carries a user-customized payload structure, like surface attenuation or photon power. This flexible approach means that the "rays" in OptiX are abstract and can be used for other concepts, for instance, photons in photon mapping. OptiX supports multiple entry points with connected ray-generation programs.

INTERSECTION programs determine if a ray intersects a piece of geometry, and if so, return useful values like the hit position and surface normal. Intersection programs can implement spheres, triangles, quads, cylinders or any other form of geometry that can be intersected with rays.

BOUNDING BOX programs return an axis-aligned bounding box of a geometric primitive. These bounding boxes are used to optimize ray tracing performance.

CLOSEST HIT programs are invoked when the ray has found its closest object in the scene. They resemble shaders in graphics pipelines and will typically implement a shading model. Closest hit programs can recursively spawn new rays to render reflections or refractions.

ANY HIT programs are invoked when it is known that a ray will intersect an object. They are useful for shadow queries, where we are concerned about occlusion but do not need to find the closest occluder. They are also useful if we want to find every object that intersect a ray, but do not care about their order.

MISS programs are invoked if a ray misses all geometry. They can be used to provide a background color or implement environment maps [Gre86].

SELECTOR VISIT programs can be used to implement more sophisticated scene traversal schemes. For example, they can be used to implement a level-of-detail system where the distance to the object determines if a coarse or detailed model is used.

EXCEPTION programs are invoked when the engine encounters an exceptional situation, like a stack overflow during ray-tracing. They are useful for debugging and error reporting purposes.

3.9.2 *OptiX runtime*

OptiX consists of a *host-side* and a *device-side* API. The host-side API provides functions to create and configure the OptiX context, define geometry and materials, create buffers and textures, and finally launch the ray-tracing. The host-side API is C-based, which makes it available for all programming languages that can bind to C libraries. There is also a convenient C++ wrapper available.

The device-side API is used by the programs to trace rays, report intersections, access buffers, and so on. Programs are simply CUDA kernels annotated with some macros, and then compiled like any other CUDA kernel. The compilation produces a Parallel Thread Execution (.PTX) file, an assembly language for a low-level parallel virtual machine. PTX provides basic mathematical operations, memory access and control flow, as well as hardware-accelerated texture accesses. PTX is defined from the perspective of a single thread, which makes it simpler to comprehend.

An important part of the runtime is the just-in-time compiler [PBD+10]. The JIT compiler takes all the programs defined by the user and performs a set of optimization passes. The optimization passes try to combine the flexibility offered by OptiX with the performance of a custom-tailored solution. For instance, it tries to make the scene graph shallower, utilize constant memory for small read-only data, convert recursion into iteration, and so on.

The compiler takes the intricate nature of GPUs into consideration. The performance of a GPU application can be significantly reduced if there are thread divergence inside warps, or if only some of the execution units are processing at the same time. Currently, a monolithic mega-kernel solution [AL09] is considered the best approach to GPU ray tracing as it minimizes kernel launch overhead. OptiX will merge different .PTX programs into a single mega-kernel and use state machines to provide the same functionality. OptiX also provides fine-grained scheduling of ray-trace launches. The runtime will load-balance ray-batches across individual execution units using queues. It can also load-balance across multiple GPUs.

As the architecture of GPUs evolves, the most efficient way to use them will probably change as well. OptiX is free to improve its runtime while providing the same pipeline to the developer, since the pipeline generally makes few assumptions about how it is implemented. For instance, the Nvidia Kepler GK110 architecture [NVI12] supports *dynamic parallelism*, where GPU threads can spawn new threads without involving the CPU. Dynamic parallelism could provide a new range of optimizations to further improve the speed of OptiX. So far, it has been apparent that the OptiX team continuously provides performance improvements with new releases.

4

IMPLEMENTATION

We implemented a renderer, which we call the Opposite Renderer[1], using C++ and the Qt Framework [Qt13]. We use the Visual Studio 2010 IDE and compiler, but it should be possible to compile the application for any system which supports Qt. Opposite Renderer is licensed as open-source project and available online for anyone to check out, test and improve [2].

The rendering engine was implemented using the OptiX framework [NVI13a], version 3.0.0. The OptiX SDK contains a minimalistic example on progressive photon mapping, which was used as a starting point for our implementation. The sample from the SDK contains two hard-coded scenes and one material type, and is not optimized for GPU. We have significantly modified and extended it in many ways.

To summarize its core functions, our Opposite Renderer contains:

- The path tracing and "memoryless" stochastic progressive photon mapping algorithms, both of which can be executed on multiple GPUs in parallel.
- Diffuse, glass, mirror and textured materials (with support for normal maps), as well as homogeneous participating media.
- Stochastic effects like anti-aliasing and depth-of-field using a thin lens camera model.
- Object-oriented structure, making it simple to extend the renderer with new materials or geometry by writing the corresponding OptiX programs.
- An user interface with status information and configurable output settings, and a viewport where the user can move the camera.
- Proper GPU random number generation using CuRAND [Nvi13d].
- Two GPU-only photon map implementations, Sorted Grid and Stochastic Hash, which offer significant performance increase over a CPU-based k-d tree implementation.

1 The name is a reference to the Seinfeld episode *The Opposite.*
2 https://github.com/apartridge

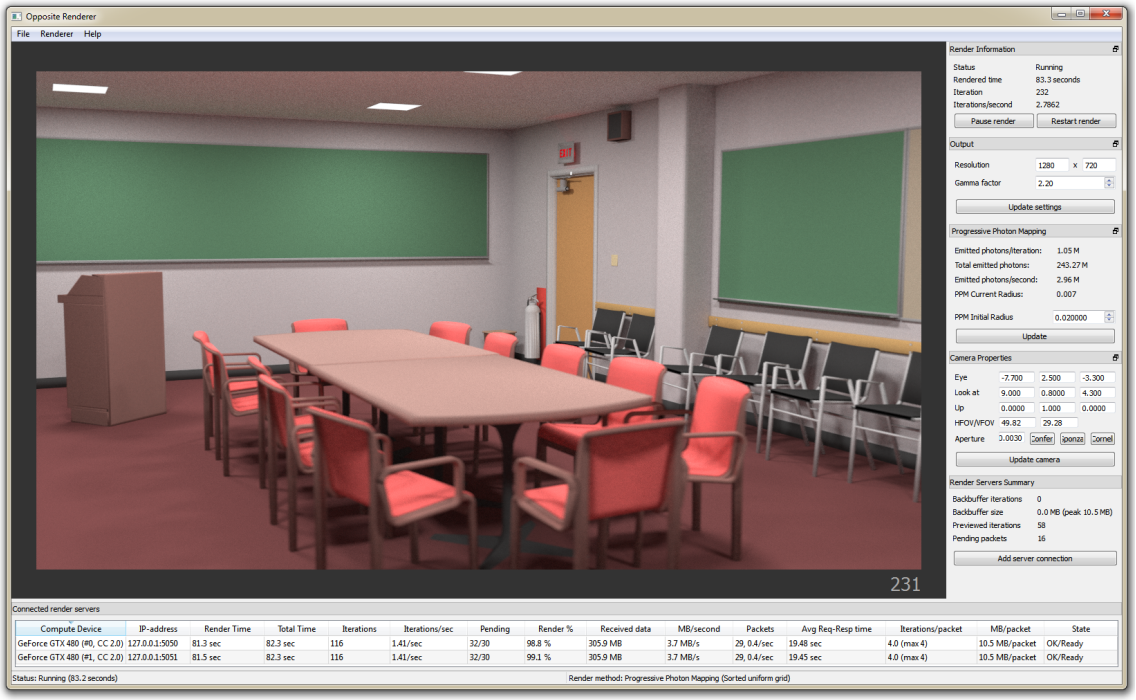

Figure 4.0.1: A screenshot of our implementation, named Opposite Renderer. The user interface displays status information and lists every connected GPU. Here, we render the Conference Room test scene using two GPUs.

- Loading of custom scenes from standard formats like Collada, 3DS and Blend. Most 3D modelers can export to these formats, including open source Blender. Geometry, material properties, lights and camera are defined in the 3D modeler and then imported. This makes our implementation available for others to experiment with.[3]

The rest of this chapter will focus on the features we implemented directly related to photon mapping. First, we describe three different implementations of the photon map. Then, we discuss how we extended the renderer to support participating media using only the GPU. Finally, we present our distributed multi-GPU implementation.

4.1 The Photon Map

The tracing, storage and subsequent gathering of photons is the most expensive part of the photon mapping algorithm. We consider three implementations of the photon map. The first option is a CPU-based *k*-d tree. The other two approaches are GPU-specialized data structures; a sorted uniform grid and a stochastic hash. In the results chapter, Sec. 5.3, we present a benchmark comparison of the three implementations and analyze their performance.

3 We use the Open Asset Import Library (assimp, `http://assimp.sourceforge.net/`) to import scenes.

4.1.1 *Kd-tree*

We start by presenting the CPU-based k-d tree implementation from the OptiX SDK [NVI13a]. A k-d tree is a specialized binary space partitioning tree which is efficient for nearest neighbor queries. Each node in the k-d tree divides the space in two along one of the axes. In a balanced k-d tree, each leaf node is the same distance from the root (plus/minus one level). A balanced k-d tree can be constructed by placing the root node at index 0, and the children of any interior node i on position $2i + 1$ and $2i + 2$. This implementation make parent and child indices implicitly available from the current node index.

The k-d tree implementation stores all traced photons in a GPU buffer during the photon tracing phase. Space for a maximum of M photon deposits for each emitted photon is reserved. Each emitted photon thread owns a part of this buffer (M elements) in which it stores interactions with non-specular surfaces. When photon tracing is completed, this buffer is transferred to the CPU. The CPU then constructs the balanced tree. It rotates between the axes (`X`, `Y` or `Z`) as it recurses down the tree, and split on the median photon along this axis. All interior nodes store the axis which the photon splits the space (using a bit flag). The leaf level of the tree is filled with specially tagged `null` nodes to simplify photon gathering.

Photon gathering is performed using a thread-local stack. We start at the root node and recurse down the side which is closest to the query position. If the splitting plane is closer than the radius r, we must push the far child on the stack so we can investigate it later. We accumulate the power of all photons within r. When we hit a leaf or `null` node, we pop the next node from the stack. Since each interior node splits the space in half, nearest neighbor search for random points is $\mathcal{O}(\mathit{height}) = \mathcal{O}(\log n)$. For typical photon map sizes, like 512^2, at least 18 lookups are required to find every photon.

The k-d tree is slow to construct on the CPU, and GPU-CPU transfer speeds add additional overhead. Any kind of tree construction does not naturally parallelize to the GPU because of the scattered nature of reads and writes. GPU implementations have been investigated, for instance by [ZHWG08], but it is not trivial and requires substantial host-device synchronization. Instead, we opted to test out two approaches which map better to the GPU, a sorted grid and a stochastic hash.

4.1.2 *Sorted Grid*

A uniform grid divides a scene into a 3D-grid of cells (*voxels*) (see Fig. 4.1.1). A photon P's index value $\mathrm{Idx}(P)$ is its linear index in this 3D grid. By sorting the photons based on their index value and creating an offset table, we can find the first photon in any voxel in constant time and then enumerate all photons in that cell sequentially.

The algorithm we implemented follows the description of Fleisz [Fle09], which used it for (non-progressive) photon mapping on the GPU. Fleisz named it a spatial hash, however, to be precise we choose to denote to it as a sorted grid.

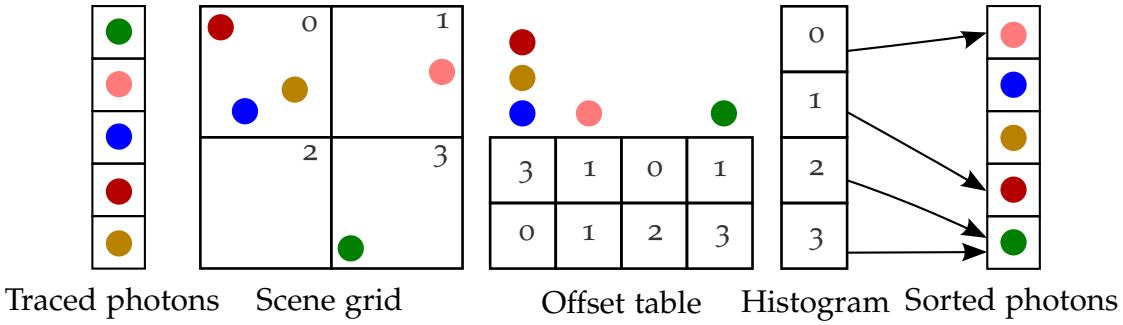

Figure 4.1.1: The SORTED GRID with five photons in a 2D-scene. The offset table points to the first photon in any cell.

The idea is not new; for instance, Purcell *et al.* [PDC+03] suggested it in 2003, several years before CUDA and mainstream GPU programming.

We use the THRUST framework [THR13] to construct the SORTED GRID. THRUST is a parallel algorithms library which resembles the C++ Standard Template Library interface, boosting useful features like sorting and partitioning. We present this approach, which resulted in a GPU only efficient and easy to implement solution, over the next subsections.

Finding photon bounding box

The first step is to trace photons and store them in a photon array. This step is identical to the K-D TREE. The initial step of the construction algorithm is to find the axis-aligned bounding box (`AABB`) of all photons, so we can calculate grid indices. The bounding box can be found efficiently using `transform_reduce` in THRUST. First, each photon is transformed to an `AABB` that contains only that photon. In the reduce step, two `AABB`s are combined to a single `AABB` that encapsulates both. The result of the reduction is an `AABB` that encapsulates the entire scene. While it is certainly possible to use the scene's `AABB`, this approach adopts better to the situation if photons are deposited in a part of the scene. The reduction operation was observed to take only a small percentage of the total construction time.

Calculating indices for each photon

With the photon `AABB`, we can calculate grid indices for each photon. We can find the 3D grid index of any point P inside the grid as $\mathrm{G}(P) = \lfloor (P - worldOrigo)/cellSize \rfloor$. The subtraction of *worldOrigo* normalizes the extent of the photons. The 1D index is

$$\mathrm{Idx}(P) = \mathrm{G}(P).x + \mathrm{G}(P).y \cdot GridSize.x + \mathrm{G}(P).z \cdot GridSize.x \cdot GridSize.y$$

The number of distinct voxels is $N = GridSize.x \cdot GridSize.y \cdot GridSize.z$, and the index values range from 0 to $N - 1$. We calculate the grid index for each photon using a simple CUDA kernel with one thread per photon.

Sorting the photons

With the one-dimensional grid index for each photon, we need to sort the photons by this value. We store the grid index for each photon in a separate array; therefore we use THRUST's `sort_by_key`. We found this to be faster than storing the hash value as a part of the photon structure and sorting photons directly. However, this is dependent on hardware and also the size of each photon. Since some of the photons in the photon buffer are invalid (they may have missed the scene entirely), we move invalid photons to the very end of the photon array where they can be forgotten.

Offset Table

At this time, we have the photons sorted by their cell. The last required piece is the offset table. This table gives the offset of the first photon in any cell. First, we produce a histogram of grid indices; a count of the number of photons that map to each cell. The simplest approach is to do an atomic addition to a histogram array when we calculate indices (Sec. 4.1.2). This prevents another pass over the photons. This approach has good performance since the photons are distributed among many cells. If many photons map to a few cells, a serialization penalty can incur.

With a histogram of the grid indices, the offset table is constructed with a *prefix sum* (cumulative sum) operation. In THRUST, we use the `exclusive_scan` function. We construct the offset table with an added entry at the end, where we store the total number of photons. This is necessary so that we can find the total number of photons in the last cell during photon gathering.

Photon Gathering using the Sorted Grid

With the sorted photon array and the offset table, we can find the first photon in any cell in constant time, and then enumerate every photon in that cell sequentially. We can get the number of photons in any cell by finding its offset in the photon array, using the offset table, and subtracting that from the offset of the *next* cell. Listing 4.1 explains this in code.

During radiance estimation, we need to find all photons within a distance r of a query point p. We are required to find all cells that intersect a sphere of radius r placed at p (Fig. 4.1.2). We consider each axis in turn. For the x axis, we can find *xMin* and *xMax* as

$$\begin{aligned} xMin &= \max(0, \lfloor (p.x - worldOrigo.x - radius)/cellSize \rfloor) \\ xMax &= \min(GridSize.x - 1, \lfloor (p.x - worldOrigo.x + radius)/cellSize \rfloor) \end{aligned}$$

Similarly for y and z. Now we have intervals (*xMin*, *xMax*), (*yMin*, *yMax*) and (*zMin*, *zMax*) of the grid we need to check for photons. We can exploit the fact that for sequential $x's$, $\mathrm{Idx}(P)$ is sequential, and their photons are sequential in the photon array. This removes one nested loop.

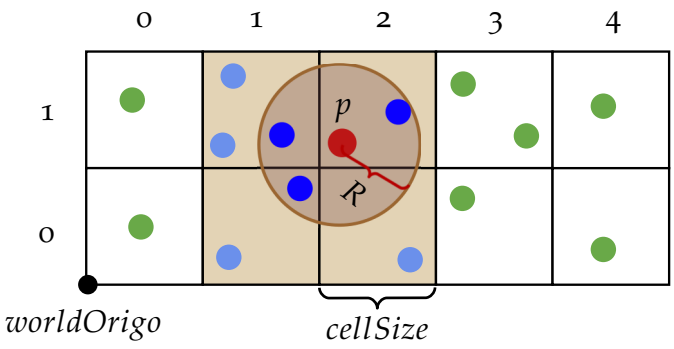

Figure 4.1.2: The SORTED GRID investigates every cell which intersects a sphere of radius R around p.

Listing 4.1 Finding photons in a radius R of a query point p using the SORTED GRID.

```
normPosition := p - worldOrigo;
xMin := max(0, floor(normPosition.x - R)/cellSize));
xMax := min(GridSize.x-1, floor(normPosition.x + R)/cellSize));
yMin := max(0, floor(normPosition.y - R)/cellSize));
yMax := min(GridSize.y-1, floor(normPosition.y + R)/cellSize));
zMin := max(0, floor(normPosition.z - R)/cellSize));
zMax := min(GridSize.z-1, floor(normPosition.z + R)/cellSize));

getHashValue(x, y, z){
    return x + y*gridSize.x + z*gridSize.x*gridSize.y;
}

if(xMin <= xMax){
    for(y := yMin; y <= yMax; y++){
        for(z := zMin; z <= zMax; z++){
            hashFrom := getHashValue(xMin, y, z);
            hashTo := hashFrom + (xMax-xMin);
            offsetFrom := offsetTable[hashFrom];
            offsetTo := offsetTable[hashTo+1];

            for(i := offsetFrom; i < offsetTo; i++){
                photon := photons[i];
                if(length(photon.position-p) <= R){
                    // Accumulate Power of Photon i
                }
            }
        }
    }
}
```

The main difference between our approach and Fleisz [Fleo9] is that we do not enforce a relationship between the size of each grid cell and r, the query radius. If $cellSize = r$, we must enumerate exactly $3^3 = 27$ voxels; the voxel that contains p and each neighboring voxel in every dimension. We observed that our dynamic approach gives better results on our test scenes and hardware. While we may evaluate more photons due to larger cells, we may also get away with checking fewer voxels.

Larger scenes where the camera points at a small part would require a denser grid. It is a good idea to set the cell size to some function $f(R)$, since R is configured by the user and is scene-dependent. Further investigation is required to get a better understanding of the impact of the cell size and how it should be related to R. If memory space is limited, it is possible to use a hash function instead of linear grid index $\mathrm{Idx}(P)$. Each unique value requires an unsigned integer (4 byte). We typically use 100^3 to 150^3 voxels in our test scenes, which is 4MB to 13MB of overhead memory.

An important observation is that the UNIFORM GRID approach used naively will introduce bias. Since each emitted photon tracing thread has some maximum limit of deposited photons M, there is a possibility that we may run out of space for photon deposits. In our test scenes, we usually use $M = 4$, so any photon which bounces more than five times will be dropped from consideration. Since only a fraction of the photon power is left after 5 surface interactions on our scenes, the dropped photons contribute little to the overall image. But this is very scene dependent - a closed room with bright surfaces would require a larger limit. Russian roulette can be useful to stop infinite paths and keep the photons at about the same power. One possible unbiased solution is to maintain a small overrun buffer with an atomic counter, and write to this buffer when we reach the limit. This overrun buffer will then have to be considered during radiance estimation. While it is important to stay below M deposits per emitted photons, we do want to keep M as small as possible, since each extra photon requires space and will have to be handled later.

4.1.3 *Stochastic Hash*

The second photon map we implemented is a STOCHASTIC HASH based on ideas presented by Hachisuka and Wann Jensen [HJ10]. It shares some similarities with the SORTED GRID; we divide the scene into a Cartesian grid of voxels. The difference is that we stochastically store a *single photon* per voxel in a separate hash table (Figure 4.1.3). This effectively removes the need for any post-processing (sorting) of photons.

During photon tracing, we calculate the hash value for the photon directly, running the grid index through a hash function. We then store the photon into the hash table directly. Every photon in the same voxel will have the same hash value. In practice, many photons belong to the same voxel, and therefore map to the same hash table entry. Under the assumption that photon tracing is a *random* process, the authors suggest we can write to the photon map without synchronization. In the end, a random photon has survived. This is why this

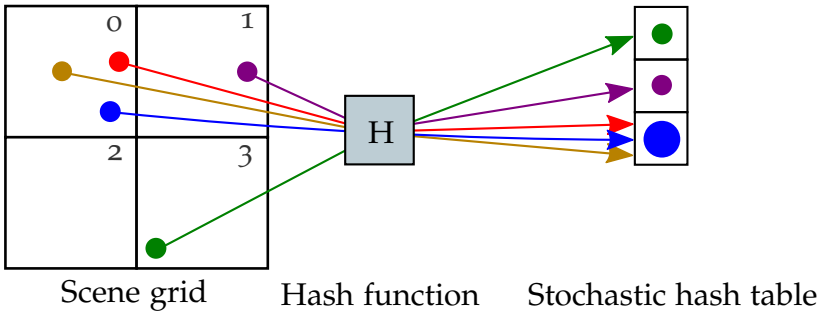

Figure 4.1.3: STOCHASTIC HASH. The grid index is passed through a hash function to find the hash table index. When multiple photons map to a single index, a random photon survives and the power is scaled by the number of collisions.

approach is named a *stochastic* hash. In order to remain consistent, we must keep a count of how many photons n map to each hash table entry. The power of the photon is then scaled by n. We can find n with an atomic increment operation. To simplify the gathering process, we let the side length of each voxel equal the radius of the photon gathering.

The main difference between STOCHASTIC HASH and the SORTED GRID is that the STOCHASTIC HASH stores a single photon in each hash cell. We do not have to sort any photons, since we write directly to its correct position in the hash table during tracing. There is no added space requirement beyond the hash table. The size of the hash table can be set to the number of emitted photons or 2-4 times larger to reduce collisions. It is advantageous for the hash table size to be a power of 2 to avoid an expensive modulo in the hash function. The need for the offset table is removed, since the photon stored in a voxel can be found directly in a single lookup. STOCHASTIC HASH increase variance, like any Russian roulette technique, since most of the photons are simply ignored. This introduces some noise to the final image.

4.2 PARTICIPATING MEDIA

This section will describe how our GPU renderer was extended to support participating media (Sec. 3.7). The beam radiance estimate [JNSJ11] (Sec. 3.7.3) was chosen because of its simplicity: it requires few extensions to our existing framework and OPTIX can do much of the heavy lifting. Our implementation currently supports heterogeneous media, where the properties of the medium are constant.

A photon in a participating medium may be absorbed, it could change direction, or it may pass unaffected trough. A medium with a high extinction coefficient ($\sigma_t = \sigma_s + \sigma_a$) has increased probability that the photon is affected by an event. The average propagation distance in the medium before an event can be found as [JNSJ11]

$$t_E = \frac{-log(\zeta)}{\sigma_t} \tag{4.2.1}$$

where ζ is a random number between 0 and 1.

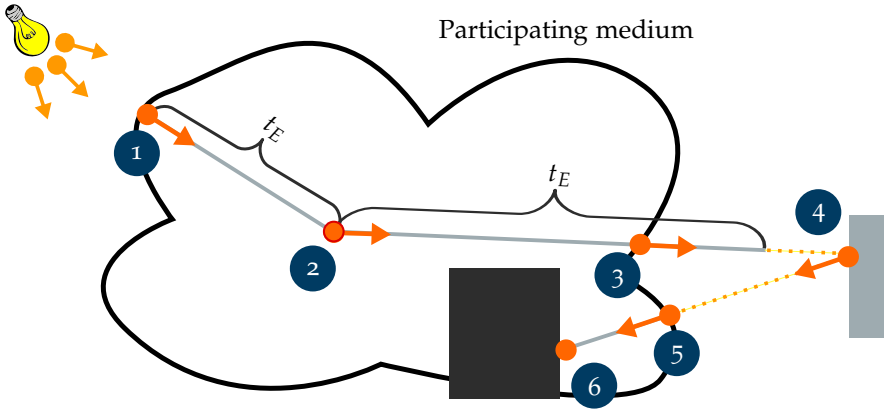

Figure 4.2.1: The life of a photon in a participating medium. 1. Photon enters medium and an event distance t_E is sampled. 2. Nothing obstructs the photon, so it is stored in the volume photon map and scattered in a random direction. A new event distance t_E is sampled. 3. Photon leaves the medium before t_E. 4. Photon is reflected off an diffuse object. 5. Photon re-enters medium, and a third event distance is sampled. 6. The life of the photon ends when it is absorbed by an object inside the medium.

When a photon enters a participating medium at a point x (Fig. 4.2.1) we sample a random event distance t_E using (4.2.1). The photon will then continue for a maximum distance t_E. If the photon leaves the medium or collides with another object between x and $x_{t_E} = x + \omega t_E$, it is not affected by the medium. Otherwise, we let an event happen at x_{t_E}. The scatter albedo $\alpha = \sigma_s/\sigma_t$ gives the probability that the photon is scattered rather than absorbed. The photon path can be ended using Russian roulette (Sec. 3.1.3), or we may scale the photon power by α. In case of a scatter event, we store the photon in the volumetric photon map. Finally, we find the scattered direction by sampling the phase function at x_{t_E}. The photon is then redirected in the scattered direction and the same procedure happen recursively.

Each photon in the volumetric photon map is interpreted as a sphere with a radius R. We use the STOCHASTIC HASH data structure for volumetric photons. A hash function is applied to find the index. We then write into the table randomly. In order to scale photon powers correctly, we atomically increment a counter per hash cell. A stochastic approach is unbiased and the only viable option for volumetric photons due to the unpredictable nature of volume scatter events.

We model a volumetric region as an axis-aligned box (AAB). The box is connected to two closest-hit programs for photon and radiance rays (Appendix C). Since this volumetric box typically covers the entire scene, we need to support geometry inside this volumetric region. Each time a scatter event has happened at a position x, we need to sample a new event distance t_E from x (Eq. 4.2.1). Furthermore, each time a photon is reflected off a surface inside the volumetric region it should immediately "enter" the medium at $t = 0$. The intuition is that the medium is "everywhere" in the scene. Naive ray-tracing will not capture this since the ray is actually in the middle of the volumetric AAB that represents the volumetric region. If the scene is a closed room completely inside the volume, we

would never intersect the volume geometry. We need to modify the intersection program of the volumetric AAB so that it reports an immediate intersection at $t = 0$. We can then sample an event distance t_E at this point.

We found the simplest approach was to have an additional "ray in participating medium" ray-type. If the ray is of the *standard* type, the intersection program for AAB's report an intersection at $t = 0$. Otherwise, the intersection program reports the distance to the outer perimeter of the AAB (as usual). The second ray type is used when we *know* that we are inside the medium, i.e. when we have sampled a distance t_E and are tracing towards t_E to see if we hit any objects or leave the medium. Clearly, in that case we do not want to keep re-intersecting the medium at $t = 0$. Both ray types are connected to the same closest hit programs, so this adjustment is transparent for most material types.

We turn volumetric photons into spheres using OptiX's support for custom geometry. The photons are organized in a bounding volume hierarchy, which OptiX can rebuild quickly on the GPU. We use the `MedianBVH` builder, which offers a good compromise between construction time and quality. Photon spheres are kept in a separate acceleration structure. To gather volumetric radiance along any ray, we simply find every sphere that intersects that ray using regular ray tracing. Since we do not have to evaluate the photons in any particular order, we implement the beam radiance estimate (Eq. 3.7.3) in an OptiX any-hit program. If we call `rtIgnoreIntersection` in the program, OptiX enumerates every photon which intersects the ray in the most efficient order.

Since the blur in the beam radiance estimate is two-dimensional, we use a similar sequence of R_i as for surface photons; $R_{i+1} = R_i\sqrt{\frac{i+\alpha}{i+1}}$ [JNSJ11]. The initial radius R_0 can be different than surface R_0, which allow us to control the amount of blur on surfaces and in volumes independently.

4.3 PARALLEL RENDERING

After we spent time optimizing performance of progressive photon mapping on a single GPU, the next step is to utilize several GPUs. Most modern motherboards have support for 2 or 3 GPUs connected to the system over the PCI-Express bus. It would be highly desirable to utilize every GPU available on the system to speed up the rendering process. As we have seen in the background chapter (Sec. 3.6.2) the new *memoryless* PPM algorithm is well suited for parallelization, since each iteration can be performed completely independent of any other.

4.3.1 *Multiple GPUs using Nvidia OptiX*

OptiX has built-in support for multiple GPUs, and the user is able to specify which devices should be used. OptiX will schedule launches on the available GPUs and distribute the load. In fact, the details of the kernel launch are invisible to the developer, which simplifies development at the cost of absent low-level control. If multiple GPUs are enabled, buffers are stored on the host and shared between the GPUs. This will drastically slow down any multi-GPU algorithm

which is very dependent on writing and reading buffers. The PPM algorithm uses buffers for photons and acceleration structures.

Additionally, ray trace launches in OptiX are synchronous, which prevents us from starting multiple launches in parallel. Even if they were asynchronous, OptiX does not provide low-level control to set which device is to be used. Using raw CUDA *would* give us the necessary level of control; however, forfeiting OptiX's other useful features would be a major setback.

Some time and effort was spent on trying to do single-process multiple-GPUs in OptiX. One option could be to do multiple threads, where each thread manages its own OptiX context. However, OptiX is not thread-safe, so this approach is not viable.

In summary, OptiX simply does not provide enough level of control for us to use multiple GPUs efficiently.

4.3.2 *Distributed multiple-GPU rendering*

The next step is to do multi-process distributed rendering[4]. The idea is to have a single process per GPU and let multiple processes communicate using sockets. Using a multi-process approach, the GPUs could be a part of the same system, or, in fact, they could be physically separated nodes. Taking it one step further, a distributed system could be used to offload rendering to a cluster of GPUs.

4.3.3 *Architecture*

At this point, we separated our implementation into several components;

EMBEDDED is a version of the renderer which supports one GPU at the time. The render engine is *embedded* in the application, in the sense that there is no network communication overhead.

SERVER responds to requests from a CLIENT and performs rendering on a single GPU. On multi-GPU systems there is would be one SERVER process running per GPU.

CLIENT can connect to multiple servers over the network and distribute the render process. The CLIENT has a GUI with status information about each connected SERVER, and a preview of the rendered image.

These are different executables with large amounts of shared functionality, including the core OptiX-based renderer and parts of the user interface. We implement the render engine and GUI as shared libraries. Qt [Qt13] is extensively used for GUI, multi-threading and inter-process communication.

4 Here, distributed refers to parallel ray tracing over a network. Distributed ray tracing is an overloaded term which can also refer to rendering soft phenomena like anti-aliasing and depth of field.

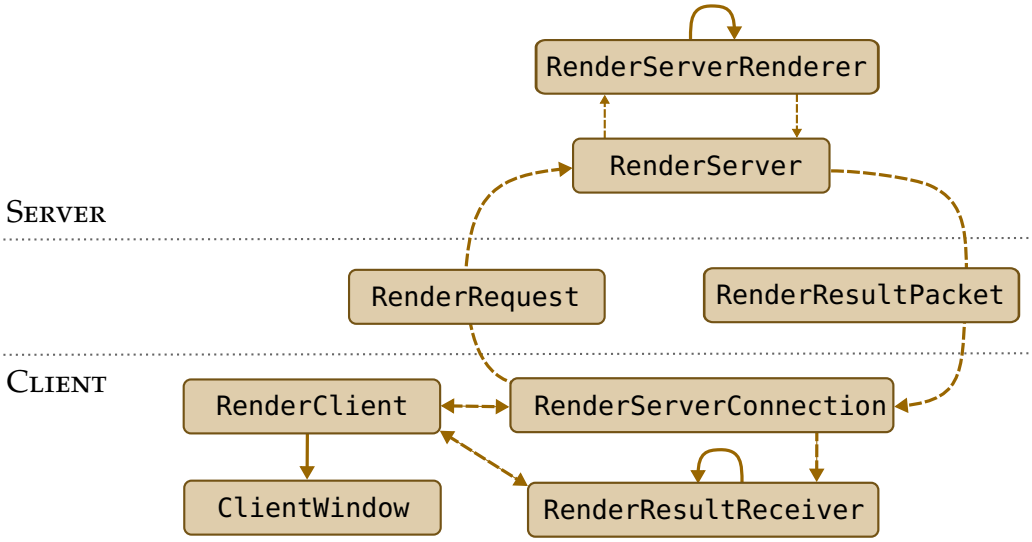

Figure 4.3.1: A client-server communication flowchart.

4.3.4 *Distributing the Progressive Photon Mapping algorithm*

We distribute progressive photon mapping (PPM) as described in Sec. 3.6.2. In summary, we can take the average of all iterations 0, 1, ..., k, where each iteration has a specified radius R_i used in the radiance estimate. The parallel algorithm has many advantages we exploit in our implementation;

- There is no dependency between iterations, so we can render them in parallel.
- We can combine several iterations into a single "packet" as long as we do a proper weighted average.

Right off the bat, some challenges present themselves;

- The client and server will need to communicate using a protocol in order to perform the rendering.
- Large amounts of data will be transferred between the entities. Network latency will pose a challenge as to maximize the performance and keep every GPU fed on work.
- When the user moves the camera or adjusts other parameters, we should take these changes into account as soon as possible.

The client can connect to a number of server processes, where each server is in control of a single GPU. On a single computer with dual GPUs, two server processes are started and configured to listen at specified ports. The client can then connect to the servers.

When the user wants to render an image, the client sends `RenderRequests` to all its connected servers. A `RenderRequest` contains the necessary data describing the rendering task: scene file name, camera properties, and image width and

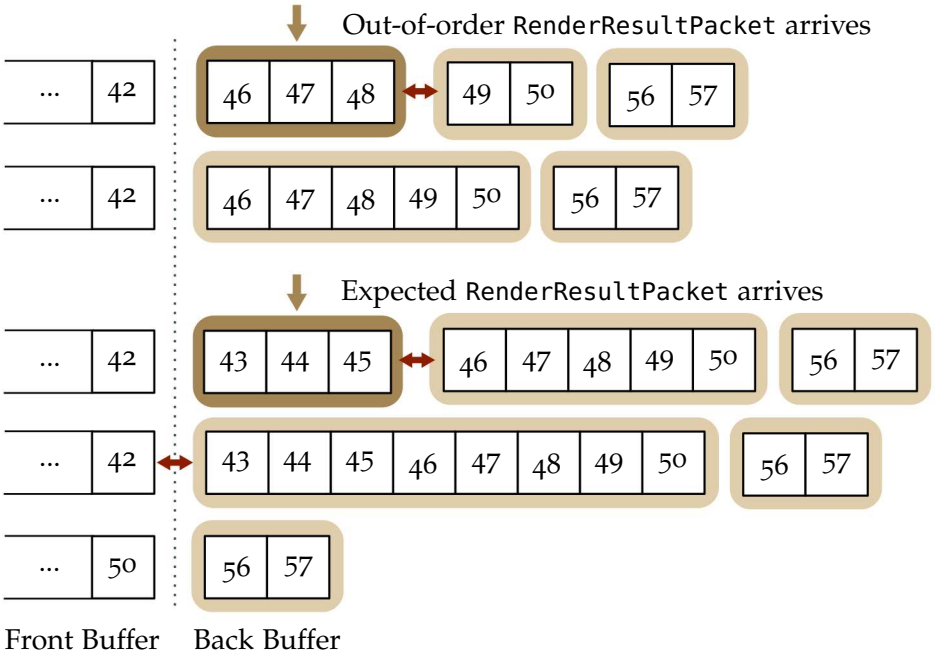

Figure 4.3.2: Arriving packets are first merged into the back buffer. If possible, the back buffer is merged with the front buffer and the iteration number forwarded.

height. The request also contains a list of N iteration numbers and their corresponding PPM radii, R_i. In a single packet, the iterations are always sequential j, $j+1$, ... , $j+N$ to simplify processing later. We use Qt's `QDataStream` to serialize and unserialize data for network transfer.

When a SERVER receives a `RenderRequest` on its network socket, it will perform the rendering procedure for each iteration in the request, using R_i as the radii in photon mapping. The packet contains a list of N iterations. We take the average rendered image of these and return the result as a `RenderResultPacket`. The size of a `RenderResultPacket` is an image-sized `float3` array plus some header bytes. For a 1280x720 pixel image, this is about 10.5 MB.

Upon reception of a `RenderResultPacket`, the CLIENT must combine this packet with others and update the rendered image. To ensure correctness of the PPM algorithm, after any iteration i the radii sequence R_0 to R_i must be used in photon gathering. Since this is a distributed computation, results are likely to come back out-of-order. Keeping two separate buffers, a front and back buffer, is necessary.

The front buffer contains the rendered image for the current iteration i. The back buffer, in a sense, represents iterations $> i$ computed ahead of time; waiting to be merged with the front buffer. Each time we receive a `RenderResultPacket`, we initially add it to the back buffer (Fig. 4.3.2). We keep the packets in the back buffer sorted on iteration numbers. First, two and two adjacent packets are compared to see if they are neighbors (consists of sequential iteration numbers). If so, we merge these two packets, taking the average weighted by the number of iterations in each packet. Finally, we try to merge the *first* packet in the back buffer with the front buffer. If the front buffer is updated, the iteration number of the algorithm is forwarded and we refresh the render on the screen.

This approach will bring iterations into the front buffer as soon as possible. Since we continuously merge packets in the back buffer, so that packets contain the longest possible sequence of iteration numbers, memory overhead is kept as low as possible. We rarely reach more than 50-60 MB of back buffer data, even after hours of rendering.

If one GPU is lagging behind, newer packets from other GPUs will fill up in the back buffer. When the missing iterations finally show up, a leap in iteration number can be observed. The consequence is that the time between updated previews, as well as performance indicators, will vary. When rendering detailed and complicated scenes, it is not a problem; we want a progressive update at regular intervals, for instance, at least every five seconds, so we can evaluate the quality and optionally stop.

To maximize GPU efficiency, they should be fed on work at all times, even as requests and results are in transit. We let the client send multiple `RenderRequests` to each server. Upon reception, the server pushes these requests to a queue. A dedicated render thread pulls new requests and executes them as they arrive. The client keeps on sending new requests for every response, to keep the server queue non-empty.

Each time a setting is adjusted or the camera is moved, we need to restart the render process. We keep a *sequence number* which we increment each time we restart. The sequence number is attached to every `RenderRequest` so the packet can be properly identified. When we move the camera, change the scene, or do other adjustments, the image should be updated as soon as possible. Since each server can have a long queue of pending requests, we need to "flush the pipe" and drop all iterations which belong to an outdated sequence number. When the sequence number is incremented, we immediately send `RenderRequests` to each server, to notify them about the change. If a server receives a request with an updated sequence number, it will forget old requests as soon as possible. Since rendering a packet may take many seconds, we check if the current packet has gone stale before iteration execution. Still, it will take some time for these messages to reach the servers and then come back. Unfortunately, there is a noticeable latency before the new image is rendered. A possible optimization is to use a local GPU to render the first frames of a new sequence.

5

RESULTS AND ANALYSIS

This chapter will present results we discovered during this thesis. Rendered images of our set of benchmark scenes are presented. The performance of the three different photon map implementations we support is analyzed. Subsequently, we benchmark our scenes on three graphics cards, including the recent NVIDIA TESLA K20. Finally, the performance increase offered by our distributed, multi-GPU implementation is investigated.

5.1 OUR TEST SCENES

We test our implementation on four different scenes. The purpose of the scenes is to demonstrate *global illumination* effects; light reaching areas and corners of the scenes which are difficult to capture. The last scene contains a participating medium which creates some very interesting volumetric effects. Several scenes are employed to stress our implementation under varying conditions. Table 3 contains scene statistics and parameters.

CORNELL BOX The CORNELL BOX (Fig. 5.1.1) was first created by Donald Goldberg and students at Cornell, and presented in a paper on diffuse reflections [GTGB84]. It consists of an open box with two colored side walls. Boxes and/or spheres inside the box demonstrate inter-reflections. The simplicity of the CORNELL BOX has made it the most widely adopted test scene for rendering applications. We created our own version of the box with a number of different sized cubes, a mirror sphere, and a glass sphere.

CONFERENCE ROOM The CONFERENCE ROOM (Fig. 5.1.2) is another classical scene, modeled after a real conference room at Lawrence Berkeley National Laboratory. The original model was created by Anat Grynberg and Greg Ward circa 1991. We use a version remodeled by Kenzie Lamar[1] with some adjustments (fixed inwards-facing normals and moved some geometry). The CONFERENCE ROOM is a medium-sized scene with Lambertian (diffuse) surfaces only. The room is lit from eight area light sources on the ceiling, as well as a red point light behind the exit sign.

1 Model downloaded from the Morgan McGuire's Computer Graphics Archive, `http://graphics.cs.williams.edu/data`.

SCENE	TRIANGLES	PPM RADIUS R_0	RESOLUTION	σ_s	σ_a
CORNELL BOX	17 600	0.02	1024×768	0	0
CONFERENCE ROOM	324 000	0.02	1280×720	0	0
SPONZA	408 000	0.08	1280×720	0	0
DISCO ROOM	70 840	0.033	1280×720	0.05	0.01

Table 3: Scene information and parameters used for the presented rendered images.

SPONZA Our third scene is a model of the Sponza Palace in Dubrovnik (Figure 5.1.3). This scene was initially created by Marko Dabrovic as an entry in a rendering contest. Over the years it has reached widespread adoption by the community. We use a version of the scene created by Frank Meinl at Crytek [2]. Meinl's version consists of more textures and geometry, making it more colorful and interesting. We illuminate the Sponza Atrium by a distant point light source to approximate the sun. It is a relatively large scene, so many photons are needed to illuminate its darkest corners.

DISCO ROOM Our last scene, DISCO ROOM (Fig. 5.1.4), is the only one in this set which demonstrates a participating media. We modeled a simple room with an open back wall. The room is illuminated from 5 colored spot lights hanging from the ceiling. The entire room is covered by a fog, and light scattering between lights and the fog is particularly visible against the black background. The fog is approximated as a diffuse, homogeneous medium covering the room. A spot light focused on a glass sphere create an interesting volumetric caustics effect. This is one example of a scene which is very difficult to render with unbiased methods; however, with photon mapping and beam radiance estimate we get nice results rather quickly.

5.2 TEST BED

We test our system on up to three computers simultaneously. Our primary pair is identical; they have a MSI Z77A-G45 motherboard with an Intel I7 3770 Ivy Bridge 3.40 GhZ processor, 32GB of RAM and a 1200 Watt power-supply. The third machine has 16GB of RAM and an 850 Watt power supply. In all cases we test on the Windows 7 64-bit OS. The computers are connected together via a gigabit Ethernet switch.

Three different graphics cards are employed in our benchmarks. NVIDIA TESLA C2070 and the NVIDIA GEFORCE GTX 480 were released in 2010. NVIDIA TESLA K20 was released in the end of 2012. Specifications for these cards are listed in Table 4.

2 Model released to the public by Crytek. `http://www.crytek.com/cryengine/cryengine3/downloads`

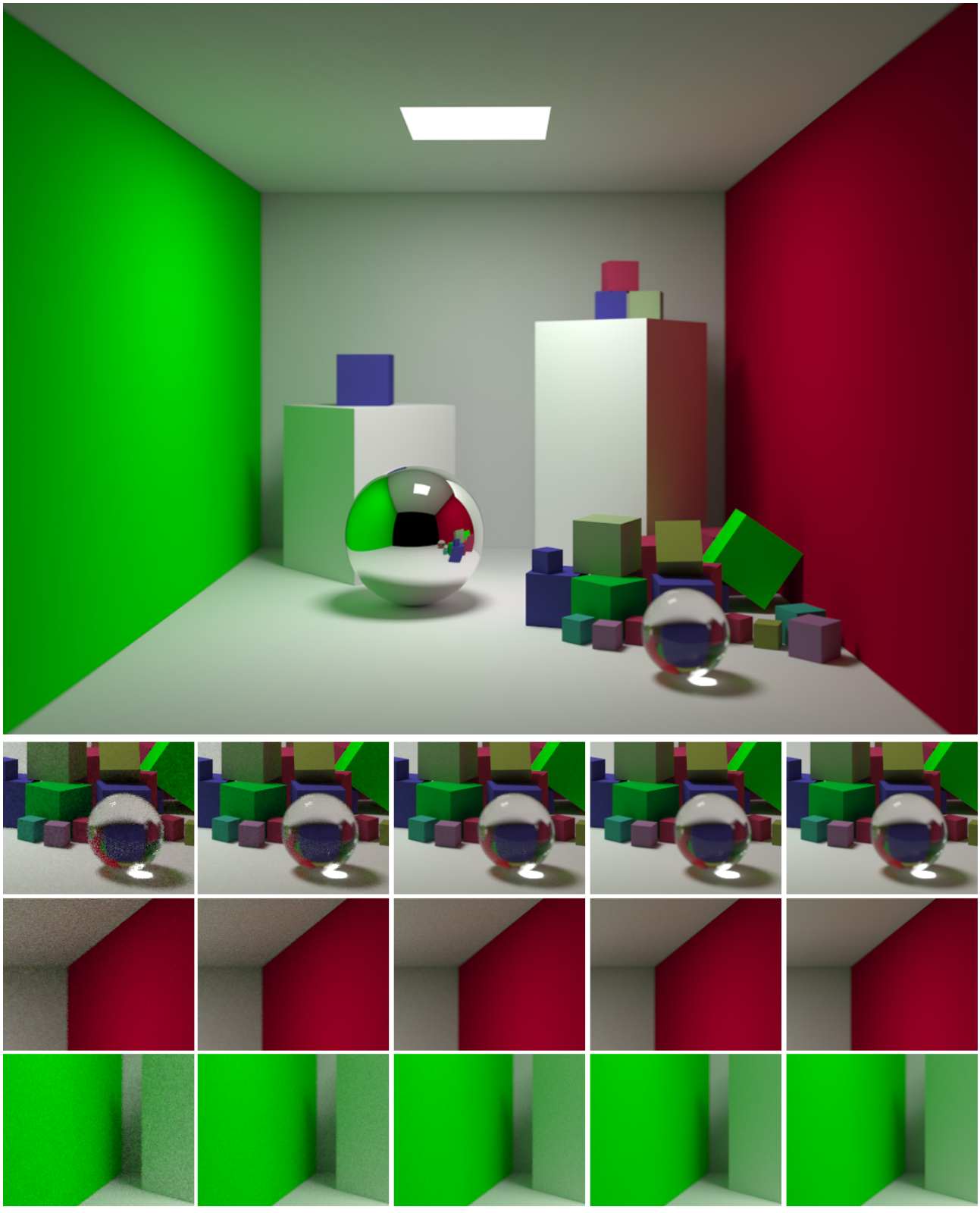

Figure 5.1.1: Cornell Box. Rendered with slight depth-of-field. Large image: *a reference image* to demonstrate a noise-free result, taken after 70 500 iterations (using 2 GTX 480 for 3h43m). With 1024^2 emitted photons per iteration, a total of 74 billion photons have been emitted. Small images, from left to right: 10 iterations, 50 iterations, 200 iterations, 1000 iterations, and reference. After 1000 iterations the overall image is of high quality, albeit some noise can be observed up close. We render 1000 iterations using progressive photon mapping in less than 1 minute on 6 Nvidia GTX 480's concurrently.

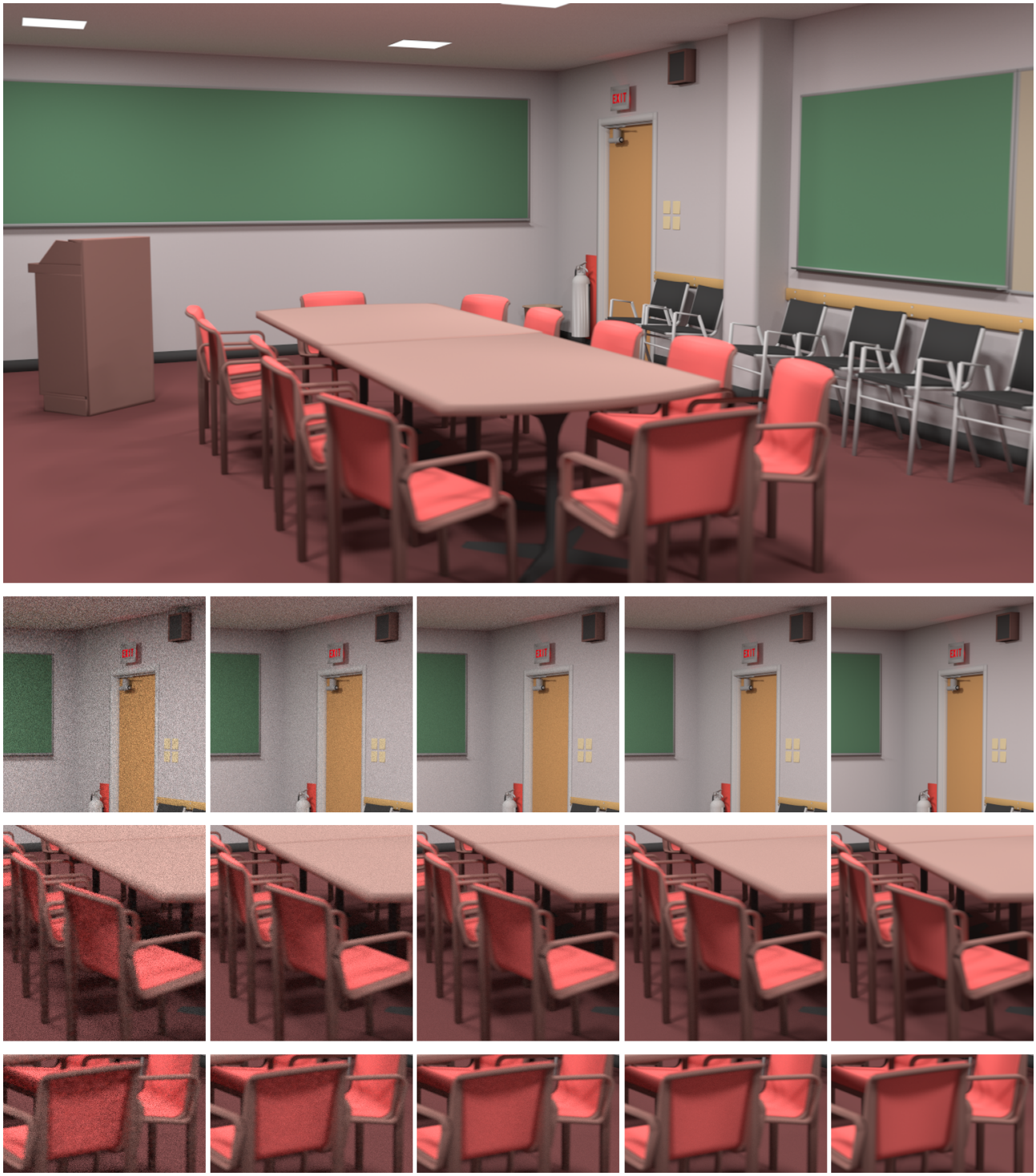

Figure 5.1.2: Conference Room. This scene is modeled using diffuse materials exclusively. Eight square light sources on the ceiling produce realistic interior lighting and multiple shadows. The closest chairs are slightly out of focus. Large image: taken after 45 500 iterations of progressive photon mapping (a total of 47 billion emitted photons), using 2 GTX 480 for 4 hours. Small images, from left to right: 10 iterations, 50 iterations, 200 iterations, 1000 iterations, and reference. Even after 1000 iterations, noise is still prominent, especially on the ceiling. A few thousand iterations are required to get close to noise-free results. We render 1000 iterations in about 1 minute 45 seconds on six Nvidia GTX 480.

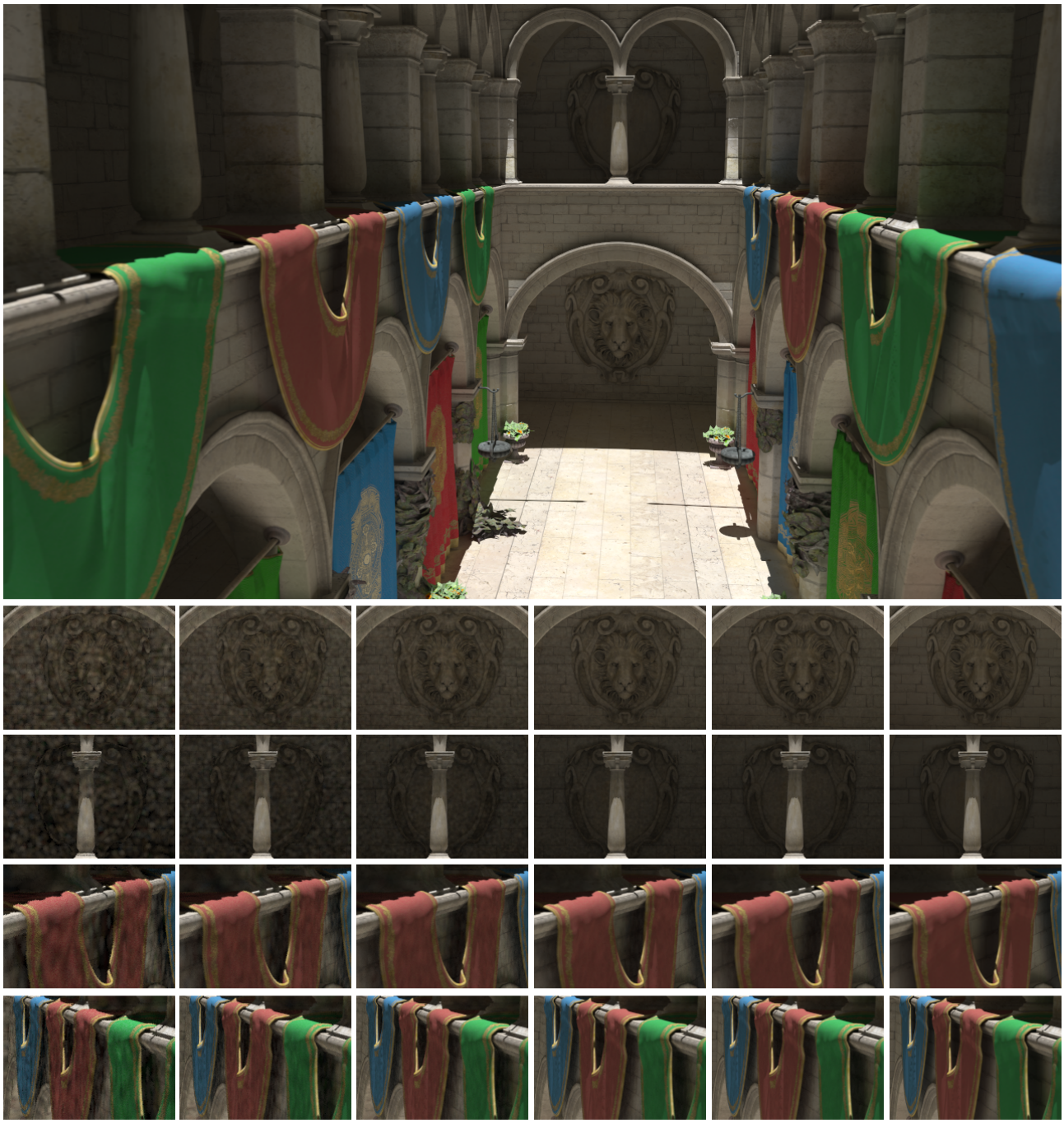

Figure 5.1.3: Sponza. Large image: taken after 353 000 iterations (370 billion emitted photons), using 2 Nvidia GTX 480 for almost 18 hours. Small images, from left to right: 10 iterations, 50 iterations, 200 iterations, 1000 iterations, 2000 iterations, and 353 000 iterations. Notice the direct sunlight burning on the courtyard, reflecting and reaching the interior hallways. Also, appreciate the soft shadows behind the colored banners, and subtle color bleeding onto the pillars. The inner hallways, including the lion face and shield, are illuminated exclusively by indirect illumination. Even after 2000 iterations, there is still noise in these challenging regions. Using 6 GTX 480's we can render approximately 1000 iterations per minute.

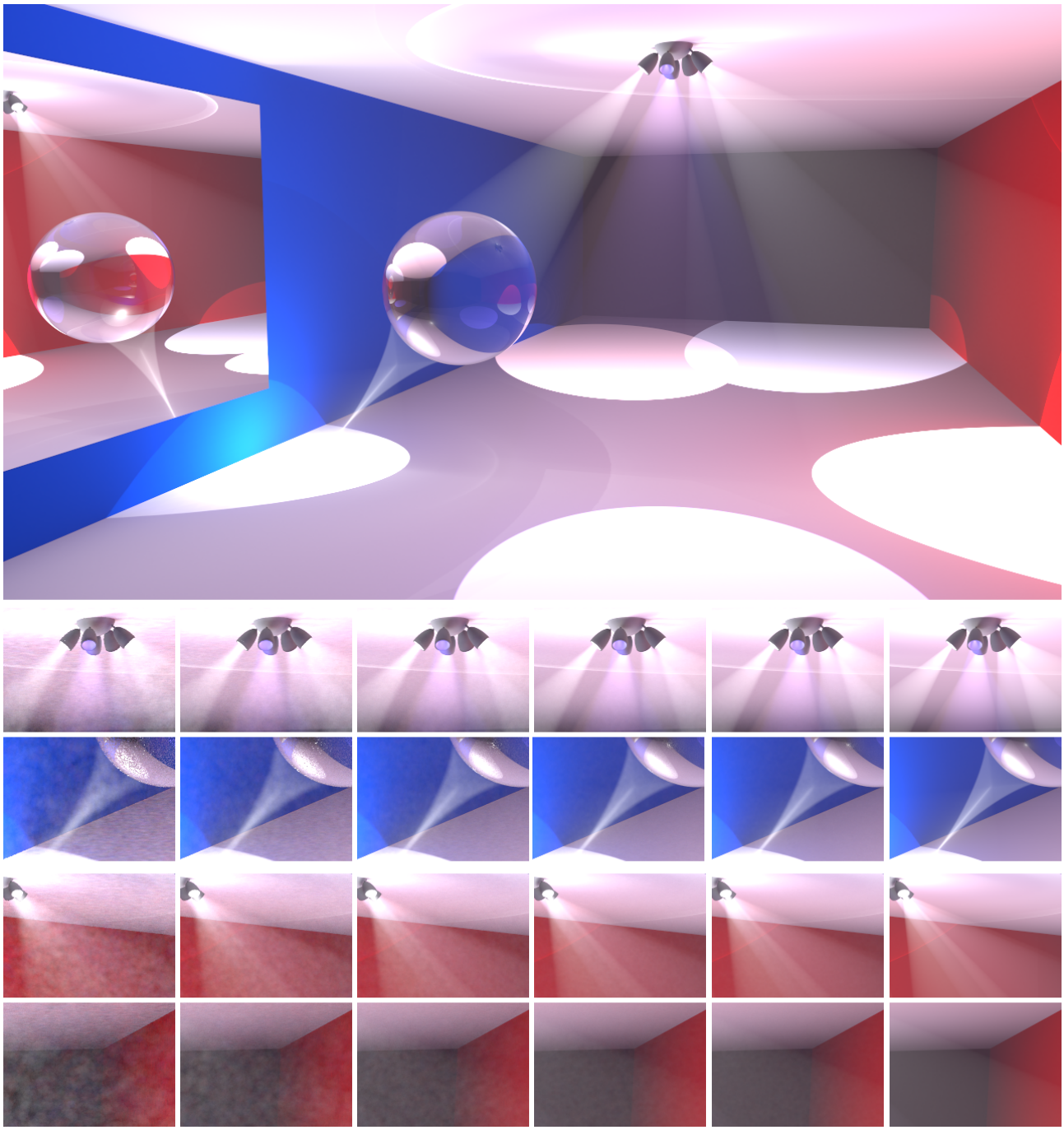

Figure 5.1.4: Disco Room. Large image: Taken after 208 100 iterations (218 billion emitted photons), using 3 Nvidia GTX 480 for almost 19 hours. Small images, from left to right: 10 iterations, 50 iterations, 200 iterations, 1000 iterations, 3000 iterations, and 208 100 iterations. This scene contains an homogeneous diffuse participating medium to model a fog. Light scattering is more intense near the light sources. Volumetric effects are particularly visible against the black background. An interesting *volume caustic* is formed under the glass sphere, reflected in the mirror as well. In early iterations of the algorithm, blur in circular patterns is a problem due to large beam radii. Even after 3000 iterations, 9 minutes of rendering with six GPUs, we have not reached razor sharp light boundaries.

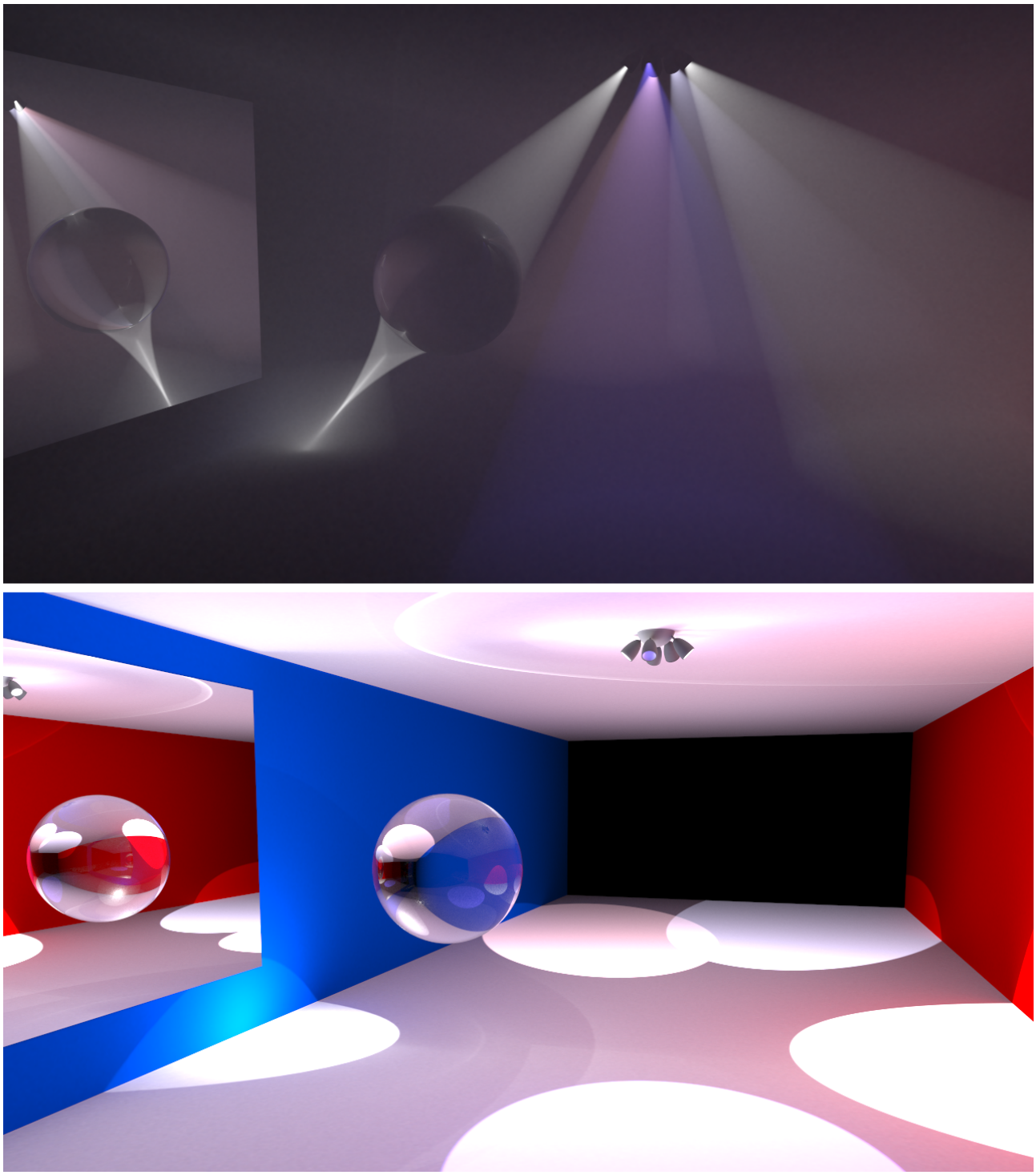

Figure 5.1.5: Disco Room. Top: volumetric radiance contribution only (12 000 iterations). The volume caustic (which is self-reflected in the glass sphere) is quite remarkable in this image. The different colored spot lights are more distinct as well. Bottom: only surface radiance. The final image (Figure 5.1.4) is the sum of these contributions.

CARD	CUDA CORES	CORE FREQ.	GLOBAL RAM	PEAK PERFORMANCE
GeForce GTX 480	448	700 MhZ	1536 MB GDDR5	1.35 TFLOPS
Tesla C2070	448	575 MhZ	6144 MB GDDR5	1.03 TFLOPS
Tesla K20	2496	705 MhZ	5120 MB GDDR5	3.52 TFLOPS

Table 4: Specifications for our test bed of graphics cards. *Source: Wikipedia list of Nvidia GPU specifications*. `http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units`.

5.3 PHOTON MAP PERFORMANCE

In a photon mapping implementation, creating and accessing the photon map will account for a large chunk of the total computation time. In this section, we compare the three photon map implementations we presented in Sec. 4.1: K-D TREE, SORTED GRID, and STOCHASTIC HASH.

This benchmark is performed on the CORNELL BOX using NVIDIA TESLA C2070. We measure the time it takes to trace photons, construct the photon map, transfer any data necessary between CPU and GPU, and finally perform the indirect radiance estimate (photon gathering). Note that ray tracing and direct radiance estimation will take a chunk of the render time per frame as well. We test with both 512^2 and 1024^2 emitted photons per iterations, as the number of photons is directly related to the time it takes to construct the photon map. For SORTED GRID and K-D TREE, each emitted photon can be stored up to four times. In SORTED GRID the scene is divided into 100^3 voxels. Measurements are done using NVIDIA VISUAL PROFILER. The reported timings are the average between iterations 200 and 204.

Fig. 5.3.1 illustrates the performance difference between the three compared photon map implementations. Table 5 contains measured timings.

Part of algorithm	K-D TREE	SORTED	STOCHASTIC	K-D TREE	SORTED	STOCHASTIC
	512^2 emitted photons/iteration			1024^2 emitted photons/iteration		
Photon tracing	90.32	89.47	88.34	351.59	351.75	347.34
Photon map construction	163.50	14.51	0.00	723.10	38.36	0.00
GPU-CPU transfers	30.52	0.00	0.00	121.39	0.00	0.00
Photon Gathering	19.33	4.36	16.41	38.45	12.59	18.02
Other	43.50	39.83	40.79	40.40	39.95	41.40
Total	347.16	148.17	145.56	1274.93	442.65	406.76

Table 5: A benchmark of photon map construction and photon gathering for K-D TREE, SORTED GRID and STOCHASTIC HASH. Timings are in milliseconds.

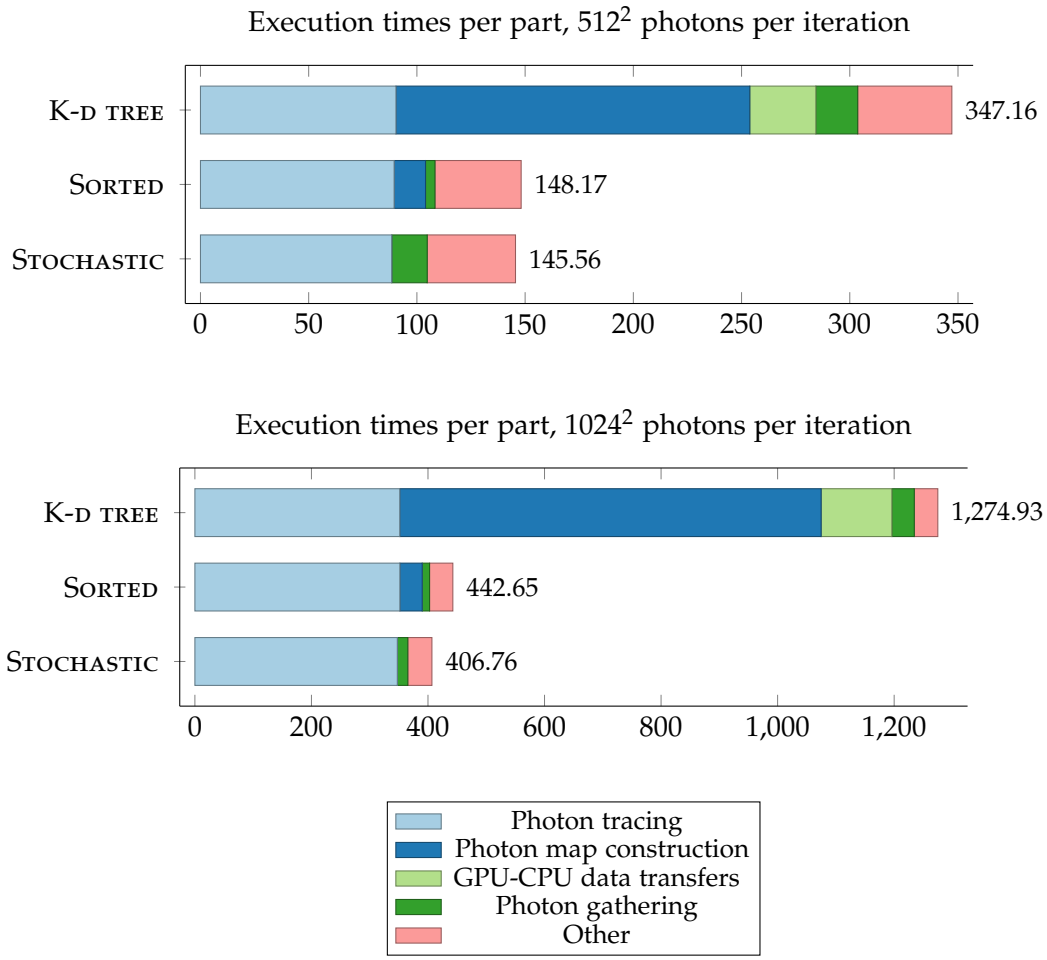

Figure 5.3.1: A stacked bar plot with timings (in milliseconds) for each part of the progressive photon mapping algorithm.

PHOTON TRACING There is (as we can observe) minimal performance difference in the photon tracing phase. STOCHASTIC HASH stores photons directly to a hash table of size equal to the number of emitted photons (512^2 and 1024^2). For the other two the photon buffer is four times as large, since each emitted photon may be deposited four times. The hash-based implementation must atomically increment a counter per hash entry, but since this is run among entries in parallel, the penalty appears to be small.

PHOTON MAP CONSTRUCTION Since STOCHASTIC HASH stores its photons directly to the hash table, the need for a construction phase is avoided entirely. SORTED GRID and K-D TREE require post-processing of photons to arrange them for gathering. Our measurements show that SORTED GRID boosts superior construction performance due to its GPU-only implementation. At 512^2 photons per iteration, it is about 13.3 times faster to construct. When we increase the photon count by 4, to 1024^2 emitted photons per iteration, the speedup increased to a factor 22.0. This is not surprising, since the construction algorithm is well-suited for

execution on the GPU. Just the transfer of photons from the GPU to the CPU and back takes more than twice as much time as the *entire* SORTED GRID algorithm.

We observe that about 78% of the time in SORTED GRID construction is spent on sorting photons. Therefore, we believe this algorithm will be accelerated further as GPU sorting performance increases. Since we use THRUST for sorting, we'll benefit from all future optimizations in its sorting kernels.

PHOTON GATHERING The performance of photon gathering is also important, and should be weighed against the time it takes to construct the photon map.

A surprising observation is that in our implementation, SORTED GRID performs best, it even outperforms STOCHASTIC HASH. We believe the main reason for this behavior is related to the size of each voxel. Since STOCHASTIC HASH uses a voxel width equal to the gather radius, it must enumerate 27 hash table entries, the neighbors in all dimensions. SORTED GRID can use a larger volume of each cell so that, on average, fewer cells must be investigated. The algorithm will enumerate every photon in every voxel that intersects the *sphere* of the photon search. Obviously, only a fraction of these photons may be within the radius and actually contribute. On the other hand, since photons in the same voxel are stored sequentially, the GPU can efficiently stream the data. It is also likely that the L1 and L2 caches on the GPU will mitigate this penalty.

In any case, K-D TREE traversal is the slowest alternative. K-D TREE gathering is based on a stack-based traversal, generally considered efficient for nearest-neighbor queries on the CPU. The number of reads is at *least* the height of the tree, and probably more. In our benchmarks, the tree is 20-22 levels high. Furthermore, these reads are spread out across the entire photon array. This pattern of scattered reads is not ideal for the GPU.

SUMMARY We have analyzed two implementations, SORTED GRID and STOCHASTIC HASH, both of which outperforms the K-D TREE based implementation from the OPTIX SDK by a landslide. These improved photon maps significantly accelerate the photon mapping algorithm. STOCHASTIC HASH has the lowest construction time; in fact, it does not require any processing after the tracing phase. SORTED GRID must sort photons, which does take some time, but the gather step is a bit faster and, most importantly, the end results suffer from less noise. Therefore, SORTED GRID is our preferred choice for surface photons. STOCHASTIC HASH is the best choice for volume photons since it can handle any number of photon deposits without introducing bias.

It may be possible to optimize both K-D TREE and STOCHASTIC GRID traversal, for instance, looking at shared or texture memory, but it may require functionality currently not available in OPTIX. We have not investigated this matter further, since we discovered that the SORTED GRID is both fast to construct and fast to traverse. The overhead of construction and gathering is only a fraction of the entire render process.

We believe the SORTED GRID is more viable with recent advances in GPU architecture, and the availability of optimized sorting libraries like THRUST. STOCHAS-

TIC HASH may suit older, more pipelined graphics cards better. STOCHASTIC HASH is more practical to implement if we are limited to graphics shaders.

We conclude from our measurements that photon tracing is a *major* bottleneck. When emitting 1024^2 photons per iteration close to 80% of the time is currently spent on tracing photons. Since each emitted photon is sent in a *random* direction from a *random* light in the scene, inter-warp coherency is low. It might be possible that OPTIX sub-optimally schedules photons across stream processors in this scenario. Unfortunately, there is no end-user control over how threads are launched, so we are prevented from investigating this matter further. We present some other ideas to improve this situation in the chapter on future work.

5.4 SINGLE GPU RENDERING

We test rendering performance on our benchmark scenes using three different graphics cards: NVIDIA GTX 480, NVIDIA TESLA C2070 and NVIDIA TESLA K20 (Table 4). We emit 512^2 and 1024^2 photons per iteration, using the SORTED GRID photon map implementation with 100^3 voxels. We use four shadow samples per pixel in the direct radiance estimate. These measurements do not include any start-up time, nor the time it takes to construct the scene, transfer geometry, build the acceleration structure, and so on. We restart the render process a couple times to prevent "cold start" bias. Measurements are done after 200 iterations of the algorithm, at which point we calculate the number of iterations per second (over the entire interval). The reported numbers are the average of four individual runs.

Absolute and relative performance for our cards is presented in Fig. 5.4.1. The TESLA C2070 performs a bit slower than GTX 480 on all our scenes. These are cards of a similar generation; however the Tesla is operating at a lower clock rate which accounts for some of this difference. Overhead related to ECC (Error Correcting Code) on the Tesla may also decrease render performance. We are not exploiting the C2070's extra memory capacity since our scenes are well within the 1.5GB limit of GTX 480.

The TESLA K20 is demonstrated to offer a speedup of 1.55x to 2.10x over TESLA C2070 on our scenes. This is less than the theoretical FLOPS increase offered by Tesla K20 (about 3.4x). It is possible that OPTIX does not take full advantage of K20's new hardware features, like dynamic parallelism [NVI12]. Since thread launches are handled by OptiX and not the developer, we are unable to experiment with other choices of grid and block dimensions.

We see that performance is closely related to the image resolution, the complexity of the scene (the number of triangles), the number of lights, and the presence of participating media. CORNELL BOX is the fastest scene to render since it has the lowest resolution and triangle count. CONFERENCE ROOM is a bit slower to render due to the number of lights (many lights decrease GPU warp efficiency in the photon tracing phase). DISCO ROOM is the slowest scene due to the participating medium effects.

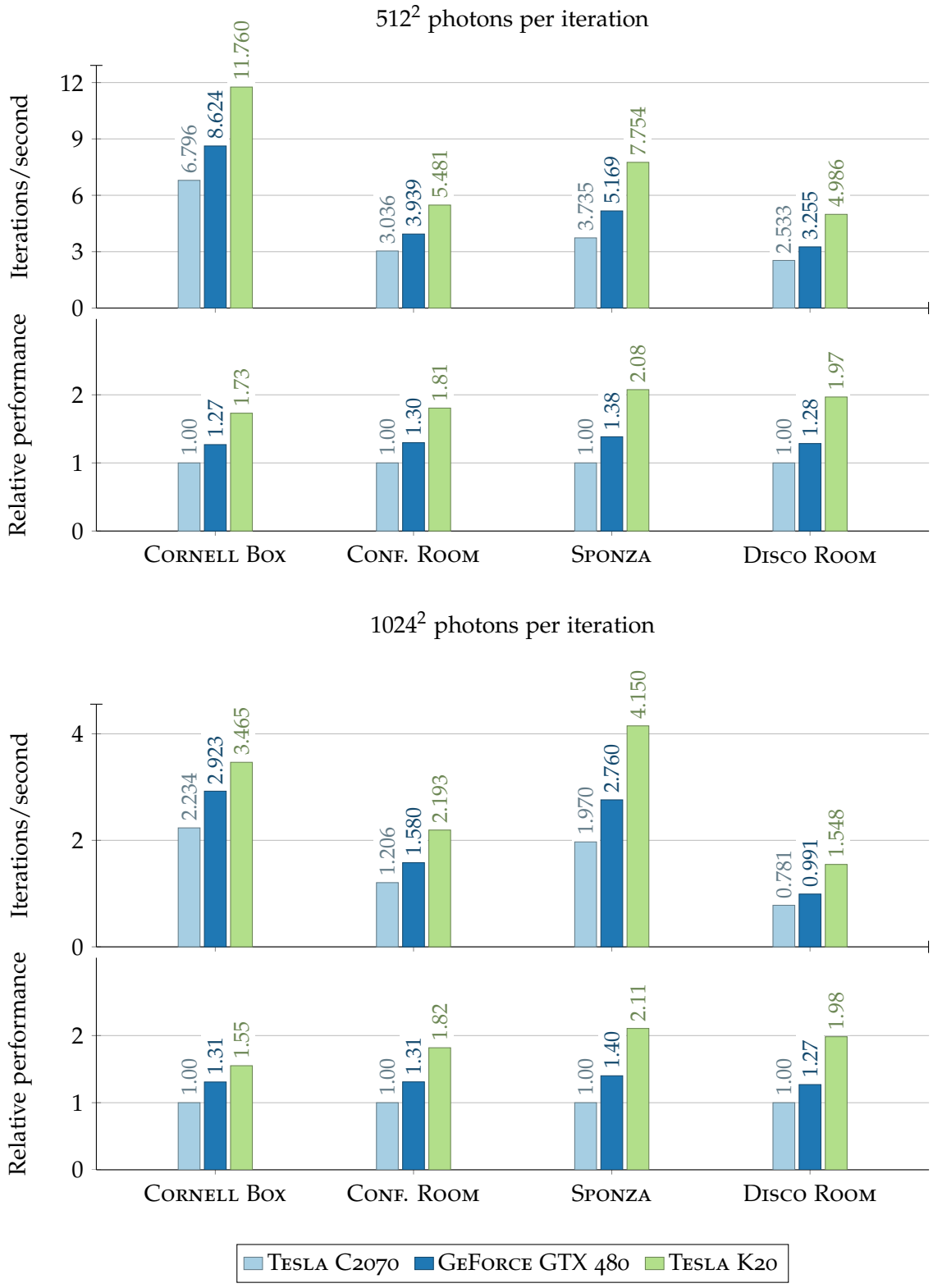

Figure 5.4.1: Graphs with absolute performance (iterations/second) in top plot, and relative performance in bottom plot.

	# GPUs	After 200 iterations			After 1000 iterations		
		Iterations/sec	Speedup	Efficiency	Iterations/sec	Speedup	Efficiency
Cornell Box	1	2.89	1.00x	1.000	2.92	1.00x	1.000
	2	5.77	1.99x	0.997	5.83	1.99x	0.997
	3	8.15	2.82x	0.940	8.56	2.93x	0.976
	$2+2=4$	10.93	3.78x	0.945	11.29	3.86x	0.966
	$2+2+2=6$	15.38	5.32x	0.886	16.84	5.76x	0.960
Conference	1	1.58	1.00x	1.000	1.59	1.00x	1.000
	2	3.13	1.98x	0.992	3.16	1.99x	0.993
	3	4.58	2.90x	0.966	4.70	2.95x	0.984
	$2+2=4$	6.04	3.82x	0.955	6.20	3.89x	0.974
	$2+2+2=6$	8.66	5.48x	0.913	9.12	5.73x	0.955
Sponza	1	2.75	1.00x	1.000	2.76	1.00x	1.000
	2	5.41	1.97x	0.985	5.48	1.99x	0.993
	3	7.78	2.83x	0.944	8.03	2.91x	0.970
	$2+2=4$	10.35	3.77x	0.942	10.84	3.93x	0.982
	$2+2+2=6$	14.29	5.21x	0.868	15.71	5.69x	0.949
Disco Room	1	0.99	1.00x	1.000	1.01	1.00x	1.000
	2	1.97	1.98x	0.992	2.01	2.00x	0.999
	3	2.83	2.85x	0.951	2.96	2.94x	0.980
	$2+2=4$	3.75	3.78x	0.946	3.92	3.89x	0.973
	$2+2+2=6$	5.41	5.46x	0.909	5.77	5.73x	0.955

Table 6: Multi-GPU render performance in iterations per second (using Nvidia GTX 480), as well as speedup and efficiency metrics on our set of benchmark scenes.

5.5 Multi-GPU rendering

Our implementation is able to execute the progressive photon mapping algorithm using several GPUs at the same time. We implemented a distributed approach using network sockets. First, we investigate the performance on a single system with three Nvidia GeForce GTX 480. We find it interesting to measure single-system performance with multiple GPUs, since this is a home-friendly and economic setup. Three GPUs running simultaneously should really put the memory bus to a stress test. The third GPU is connected to a slower PCI-Express 2.0 slot and also acts as the display adapter.

We emit 1024^2 photons per iteration and measure after 200 and 1000 iterations of the algorithm. The tests with a single GPU have the render engine embedded, while dual and triple GPU measurements use the distributed implementation. Aside from that, we use exactly the same setup and methodology as described in the previous section.

Table 6 contains measured timings. Fig. 5.5.1 is a bar plot showing achieved speedup. Our results demonstrate that we reach close to linear speedup on two and three GPUs. After 200 iterations we reach speedups of over 1.97 and 2.81. Af-

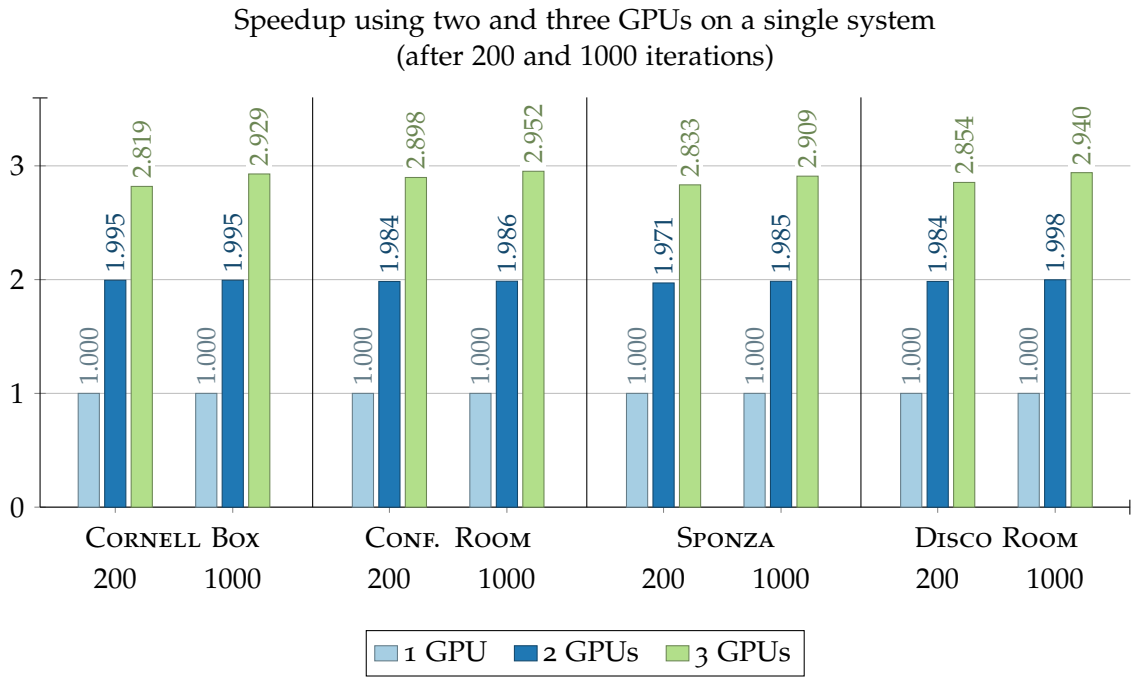

Figure 5.5.1: A bar plot illustrating speedup on a multi-GPU system. Using three GPUs simultaneously, we can present a speedup of 2.90 to 2.95 after 1000 iterations.

ter 1000 iterations we have reached better saturation of resources; we get 1.98-2.00 speedup using two GPUs, and 2.90-2.95 using three GPUs. Since 200 iterations are performed quickly with 3 GPUs, the latency of the first `RenderRequest` to the servers to get it all starting gives a slight, but noticeable latency. We also see that in the case of a single GPU, efficiency is increased from iteration 200 to iteration 1000 by about 1%, which suggest that the GPU needs some time to reach full utilization of hardware.

We believe this is as close as it is possible to get to linear speedup using three GPUs on one system. PCI-Express bus congestion, process communication delays and slightly lower performance of that third GPU will prevent us from reaching perfect efficiency.

5.6 Distributed rendering

Our presented implementation is able to distribute the rendering process on several computers using TCP/IP. This section will present and analyze multi-computer performance. We connect three computers with two Nvidia GeForce GTX 480 GPUs each using a fast gigabit switch. Two of the GPUs are at the same computer as the client, so we avoid network transfer in that case. We start four server instances on the additional pair of computers and connect to them using TCP/IP.

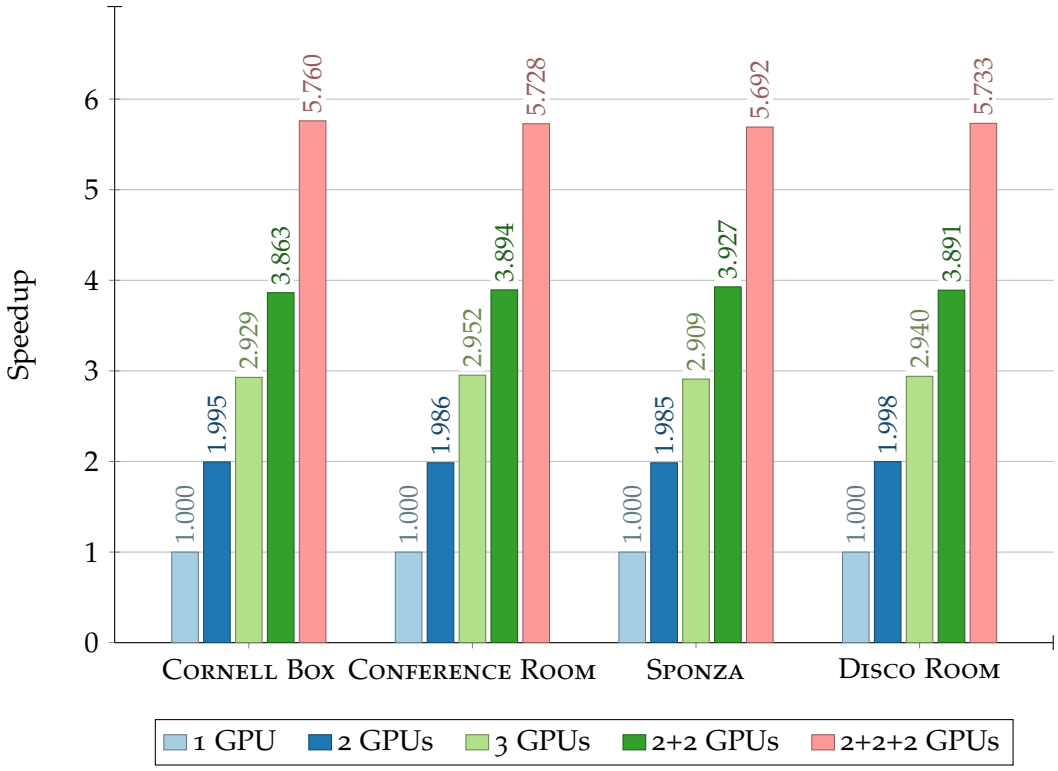

Figure 5.6.1: Speedup using up to six concurrent GPUs.

Measured timings for 200 and 100 iterations including speedup and efficiency metrics are listed in Table 6. Speedup after 1000 iterations is presented in Figure 5.6.1.

Our numbers prove that we are able to get reasonably close to ideal speedup using up to 6 GPUs concurrently. After 200 iterations, 6 GPUs can offer a speedup of 5.2-5.3. After 1000 iterations, the speedup is increased to 5.70. Since 200 iterations on 6 GPUs are executed in about 10-30 seconds on our tests, we have not reached full utilization of individual GPUs. The same phenomenon was exhibited on single-system multi-GPU rendering (previous section), however it is even more prevalent here.

We conclude that we get good speedups quickly and that multiple GPUs can accelerate the rendering of simple scenes requiring a few hundred to a thousand iterations. For complicated scenes which require thousands of iterations the efficiency is close to perfect. Feedback from the system shows that every GPU is continuously fed on work after it gets its first `RenderRequest`.

The network seems to handle this load without problems. In these tests, we reach close to 30MB/sec of data over the network, which is well within the theoretical maximum of gigabit Ethernet (~120MB/sec). Since the algorithm is trivially parallelizable, more sophisticated load balancing techniques can be implemented to handle a larger number of GPUs. For example, it is possible to adjust the number of iterations per packet if latencies are detected.

In this section, we present the timings reported by Knaus and Zwicker [KZ11] for their CORNELL BOX scene. They use a single-threaded application running on an 2.67 GhZ Intel Xeon Processor with 2 million photons per iteration. They render at a resolution of 768×768. We create a scene which matches theirs and use the same settings. They render 20 iterations of the CORNELL BOX in 459 seconds; a rate of 87200 emitted photons per second. With a single NVIDIA GTX 480 we emit $1000 \cdot 2 \cdot 10^6$ photons in 444.6 seconds, a rate of 4.49 million emitted photons per second - a speedup of 51. Using six GPUs concurrently we get another speedup of 5.70. In total, this is a speedup of over 290. While it is difficult to compare results directly, this is an indication of the level of speedup achieved using multiple GPUs over a single-threaded application on the CPU.

6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

We have described an implementation of the progressive photon mapping algorithm on the Graphics Processing Unit. A complete application based on the OptiX framework, available for others to experiment with, was presented. Several photon map variants for the GPU have been considered and analyzed. Our conclusion is that dividing the scene into a uniform grid and sorting photons by their grid index is a good approach on modern GPUs. This approach is a bit slower than a stochastic hash table-based photon map, but it produces better results, and much faster than a *k*-d tree based implementation.

Subsequently, we extended our implementation to support homogeneous participating media. We base ourselves on the beam radiance estimate, which is easy to implement with the support of the OptiX framework. We used our implementation to render an interesting image of a disco room covered in fog.

Finally, we have described recent advances of the progressive photon mapping algorithm that removed the dependency between iterations. We exploited these advances to render an image in parallel using multiple GPUs at the same time. Our described renderer is able to get near linear speedup using two and three GPUs on the same system, and even four and six GPUs connected via a fast network. Results are presented for four different test scenes exhibiting a range of scenarios. As far as we know, we present the very first implementation which is able to execute photon mapping on several GPUs at the same time. Our approach is straightforward to extend to a cluster of GPUs.

The work carried out in thesis has demonstrated that multiple GPUs can drastically cut down the execution time of the photon mapping algorithm. We believe more applications will take advantage of several GPUs to render images in the future. Cloud-based applications are particularly interesting since it gives artists and designers access to enormous amounts of power from their laptops.

We are able to render images of very good quality in a few minutes. However, minutes are not real time. The goal of implementing fog and smoke in the NTNU Snow Simulator was not achieved. For real-time simulators approximations using the depth buffer are much more efficient, although they cannot model light scattering at a *physical* level. We are able to capture effects like colored volumes, surface-volume light bleeding, and volume caustics.

There is almost an endless list of possible extensions to our renderer. To improve performance with a large number of GPUs (larger than, let's say, 8), we could introduce more sophisticated load-balancing techniques. There are numerous other improvements and functions necessary to make this renderer viable for broader use; better handling of scene files, support for more materials, scene- and image formats, better error-handling, more customizable settings, and so on.

In the current state, the photon tracing step is a *major* bottleneck. Techniques based on *stratified sampling* may increase performance. Using point-lights as an example, we could divide the sphere of directions around the light into regions, or *strata*. Every photon in the same warp could be sent through a random direction inside the same region. This would increase the possibility that they will follow a similar path in the scene for better warp efficiency. We consider this one of the most promising tracks for future performance increases.

Other optimizations at the micro-level are certainly applicable. We can compress `Photons` down to 20 bytes [Jen09] using spherical coordinates for the incoming directions, and storing powers in a shared-exponent 4-byte RGBE format. We have not spent that much time on this level of optimizations, since we were more interested in experimenting with multiple GPUs concurrently. Our work is at a higher level, so we can combine it with techniques introduced by others and increase efficiency by a solid margin.

Currently, our implementation uses the same global initial radius R_0 for every pixel. To reduce blur in the image, we can use an initial radius *per pixel*, where the radius is larger in areas with few photons, and smaller in areas with many photons. This way, the blur introduced is approximately constant with respect to pixel size [KZ11]. Initial radii R_i could be found by doing an initial k-nearest neighbor pass, or using ray differentials [Ige99].

We have implemented and demonstrated homogeneous participating media, but smoke, clouds, dust and so on are more accurately modeled as heterogeneous. Heterogeneous media is more difficult to implement since the properties of the medium varies by location. It is an interesting track for future work to consider how this could be done efficiently.

Our implementation uses the beam radiance estimate [JZJ08, JNSJ11]. A novel method based on photon *beams* rather than photon points as data have been demonstrated to produce better results faster, and is also well-suited for GPUs [JNT+11]. However, photon points could be implemented easily with support from OptiX, while beams would require more implementation. Photon beams are therefore considered future work as well.

BIBLIOGRAPHY

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149. ACM, 2009.

[App68] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968.

[Bet12] Brendan Bettinger. Pixar by the numbers - from toy story to brave. *Collider.com*, 2012. [Online; accessed 4-June-2013].

[BSS93] Philippe Blasi, Bertrand Saec, and Christophe Schlick. A rendering algorithm for discrete volume density objects. In *Computer Graphics Forum*, volume 12, pages 201–210. Wiley Online Library, 1993.

[Cha60] S. Chandrasekhar. *Radiative transfer*. Dover Books on Intermediate and Advanced Mathematics. DOVER PUBN Incorporated, 1960.

[Cha13] Chaos Software. V-Ray renderer. `http://www.chaosgroup.com/en/2/index.html`, 2013. [Online; accessed 29-April-2013].

[CHH02] N.A. Carr, J.D. Hall, and J.C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, 2002.

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

[Far11] R. Farber. *CUDA Application Design and Development*. Applications of GPU computing series. Morgan Kaufmann, 2011.

[Fle09] Martin Fleisz. Photon mapping on the gpu. *Master's Thesis*, 2009.

[FS05] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22. ACM, 2005.

[Gla13] Glare Technologies. IndigoRenderer Home Page. `http://www.indigorenderer.com/home`, 2013. [Online; accessed 08-June-2013].

[Gre86] Ned Greene. Environment mapping and other applications of world projections. *IEEE Comput. Graph. Appl.*, 6(11):21–29, November 1986.

[GTGB84] Cindy M Goral, Kenneth E Torrance, Donald P Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 213–222. ACM, 1984.

[HG41] Louis G Henyey and Jesse L Greenstein. Diffuse radiation in the galaxy. *The Astrophysical Journal*, 93:70–83, 1941.

[HJ09] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM Trans. Graph.*, 28(5):141:1–141:8, December 2009.

[HJ10] Toshiya Hachisuka and Henrik Wann Jensen. Parallel progressive photon mapping on gpus. In *ACM SIGGRAPH ASIA 2010 Sketches*, page 54. ACM, 2010.

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[Ige99] Homan Igehy. Tracing ray differentials. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 179–186. ACM Press/Addison-Wesley Publishing Co., 1999.

[Jar08] Wojciech Jarosz. *Efficient monte carlo methods for light transport in scattering media*. PhD thesis, La Jolla, CA, USA, 2008. AAI3320228.

[JC98] Henrik Wann Jensen and Per H Christensen. Efficient simulation of light transport in scences with participating media using photon maps. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 311–320. ACM, 1998.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques 96*, pages 21–30. Springer, 1996.

[Jen09] H.W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. Ak Peters Series. A K Peters, Limited, 2009.

[JNSJ11] Wojciech Jarosz, Derek Nowrouzezahrai, Iman Sadeghi, and Henrik Wann Jensen. A comprehensive theory of volumetric radiance estimation using photon points and beams. *ACM Transactions on Graphics (Presented at ACM SIGGRAPH 2011)*, 30(1):5:1–5:19, January 2011.

[JNT+11] Wojciech Jarosz, Derek Nowrouzezahrai, Robert Thomas, Peter-Pike Sloan, and Matthias Zwicker. Progressive photon beams. *ACM Trans. Graph.*, 30(6):181:1–181:12, December 2011.

[JZJ08] Wojciech Jarosz, Matthias Zwicker, and Henrik Wann Jensen. The beam radiance estimate for volumetric photon mapping. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27(2):557–566, April 2008.

[Kaj86] James T Kajiya. The rendering equation. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[KD13] Anton S. Kaplanyan and Carsten Dachsbacher. Adaptive progressive photon mapping. *ACM Trans. Graph.*, 32(2):16:1–16:13, April 2013.

[KZ11] Claude Knaus and Matthias Zwicker. Progressive photon mapping: A probabilistic approach. *ACM Trans. Graph.*, 30(3):25:1–25:13, May 2011.

[LE10] Holger Ludvigsen and Anne Cathrine Elster. Real-time ray tracing using nvidia optix. *Eurographics Short Papers*, pages 65–68, 2010.

[Lor90] Ludvig Valentin Lorenz. *Lysbevægelsen i Og Unden for en Af Plane Lysbølger Belyst Kugle*. 1890.

[Lux13] Luxrender. Luxrender Home Page. `http://www.luxrender.net/en_GB/index`, 2013. [Online; accessed 08-June-2013].

[LW93] Eric P Lafortune and Yves D Willems. Bi-directional path tracing. In *Proceedings of CompuGraphics*, volume 93, pages 145–153, 1993.

[Mie08] Gustav Mie. Beiträge zur optik trüber medien, speziell kolloidaler metallösungen. *Annalen der Physik*, 330(3):377–445, 1908.

[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, pages 77–89, New York, NY, USA, 2009. ACM.

[MLM13] Michael Mara, David Luebke, and Morgan McGuire. Toward practical real-time photon mapping: efficient gpu density estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 71–78, New York, NY, USA, 2013. ACM.

[Nvi07] Nvidia Corporation. CUDA ToolKit HomePage. `https://developer.nvidia.com/cuda-toolkit`, 2007. [Online; accessed 08-June-2013].

[Nvi09] Nvidia Corporation. Nvidia launches the world's first interactive ray tracing engine. *Nvidia.com*, 2009. [Online; accessed 08-June-2013].

[NVI11] NVIDIA Corporation. CUDA Architecture Roadmap. `http://www.nvidia.com/docs/IO/113297/ISC-Briefing-Sumit-June11-Final.pdf`, 2011. [Online; accessed 5-May-2013].

[NVI12] NVIDIA Corporation. NVIDIA Kepler GK110 White Paper. Technical report, 2012. [Online; accessed 1-May-2013].

[NVI13a] NVIDIA. NVIDIA OptiX Ray-tracing framework. https://developer.nvidia.com/optix, 2013. [Online; accessed 29-April-2013].

[NVI13b] NVIDIA Corporation. CUDA 5.0 Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2013. [Online; accessed 11-June-2013].

[NVI13c] NVIDIA Corporation. CUDA 5.0 C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2013. [Online; accessed 11-June-2013].

[Nvi13d] Nvidia Corporation. CuRAND framework. http://developer.nvidia.com/curand, 2013. [Online; accessed 09-June-2013].

[Oto13] Otoy. Octane Render Cloud Edition. http://render.otoy.com/forum/viewtopic.php?f=7&t=29544, 2013. [Online; accessed 29-April-2013].

[PBD+10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.

[PBMH02] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (TOG)*, 21(3):703–712, 2002.

[PDC+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*, pages 41–50. Eurographics Association, 2003.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Qt13] Qt. The Qt framework. http://qt.digia.com/, 2013. [Online; accessed 29-April-2013].

[Ray71] John William Strutt Baron Rayleigh. *On the scattering of light by small particles*. 1871.

[Ter11] Daniel Terdiman. New technology revs up pixar's 'cars 2'. *CNET*, 2011. [Online; accessed 1-June-2013].

[Thr13] Thrust. Thrust Framework. http://thrust.github.io/, 2013. [Online; accessed 26-May-2013].

[VG97] Eric Veach and Leonidas J Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997.

[Wal06] I. Wald. Realtime ray tracing and interactive global illumination. *IT-MUNCHEN*, 48(4):242, 2006.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 27, page 126. ACM, 2008.

A

OPPOSITE RENDERER USER GUIDE

We append a short user guide to our application. The application is available in source-code format for Windows. A 64-bit precompiled version is also available[1].

The application consists of three main entry points:

1. `Standalone.exe` supports a single GPU. In case of a multi-GPU system, the user can select the compute device initially.

2. `Server.exe` contains a server interface which clients can connect to. The user must first select compute device (Fig. A.0.2), then pick a unique port number (Fig. A.0.3). The server will start listening for client connections. When a client is connected, status information is available in the log (Fig. A.0.5).

3. `Client.exe` must be started if multiple GPUs are to be employed. This application contains the user interface where the user can connect to servers over TCP/IP (Fig. A.0.4). If the server is at the same system as the client, you should connect to it using the virtual loopback interface address (127.0.0.1) rather than public IP. If you connect to another system, we recommend using a gigabit switch to ensure that you have enough network bandwidth.

CAVEATS

The preferred tool to model scenes was BLENDER[2]. BLENDER supports COLLADA (`.dae`) files, an XML schema for digital asset exchange. COLLADA is the preferred scene format for this application since it can store camera information as well as geometry and light properties. There are currently some difficulties using textures, due to bugs in BLENDER's export. We recommend exporting to a blank folder and use the `Copy` option, which moves the textures into the same folder. At the time of writing, textures must be `.tga` image files.

Some of the settings (like the number of emitted photons per iteration, stack depth and so on) must be reconfigured in source code and then recompiled. Participating media must be hard-coded as well (since we cannot extract this information from the scene file). It is possible to write a scene source code file yourself. Look at the `scene/Cornell.cpp` file.

1 https://github.com/apartridge
2 http://www.blender.org/

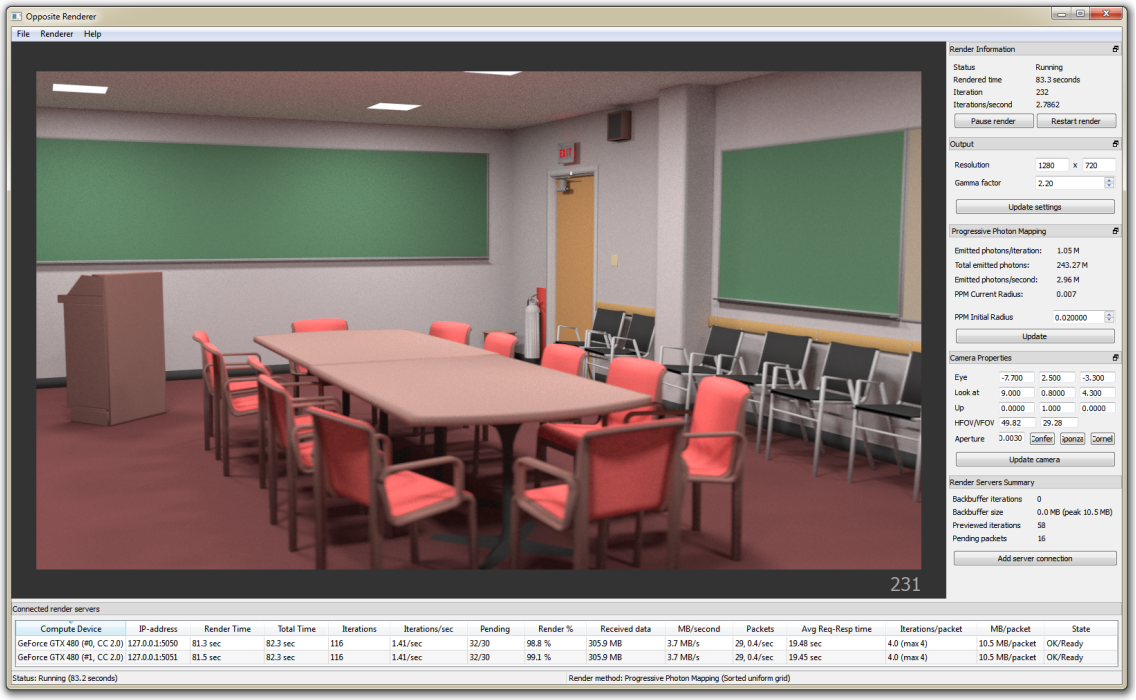

Figure A.0.1: A screenshot of the CLIENT application.

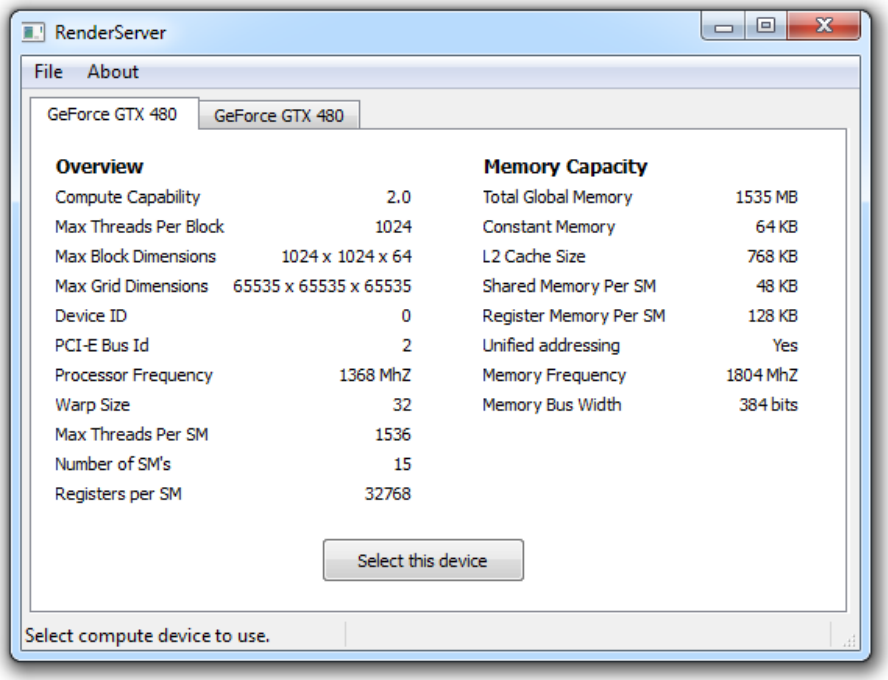

Figure A.0.2: SERVER: Selecting compute device. Various specifications for the card is listed.

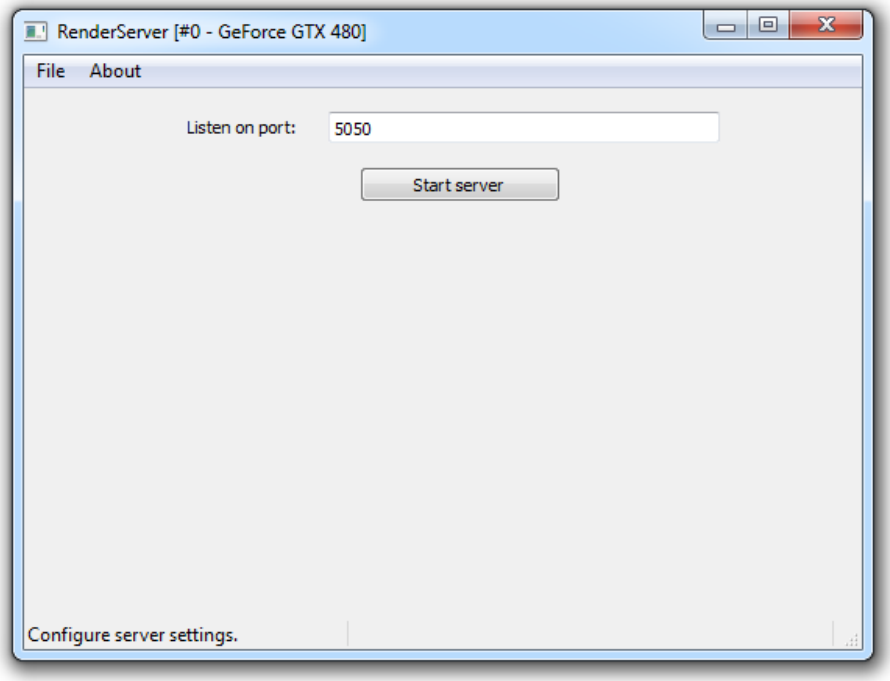

Figure A.0.3: SERVER: Set which port to listen to.

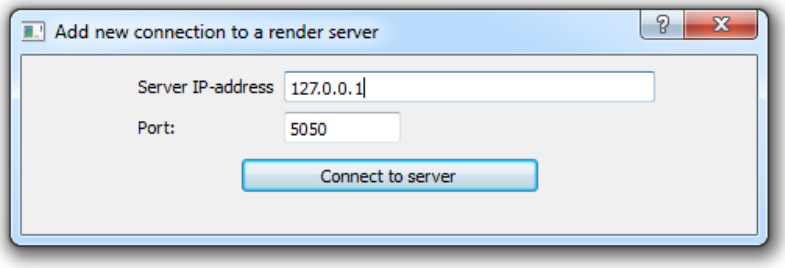

Figure A.0.4: CLIENT: Connect to a SERVER using its IP and port.

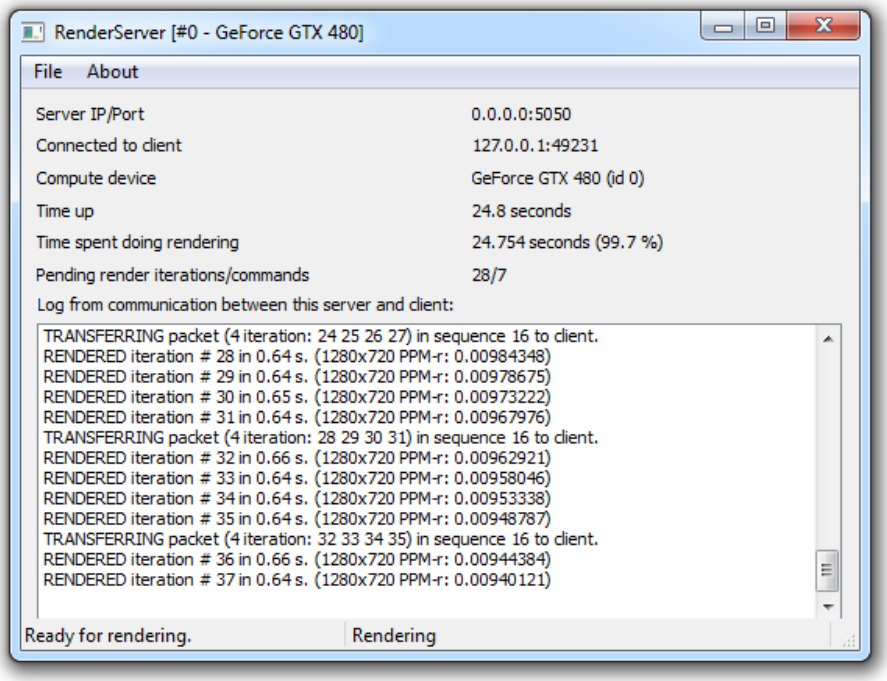

Figure A.0.5: SERVER: When connected to a client, a list of log entries is visible.

B

DIFFUSE SHADER

We include an example of a diffuse shader in OptiX (`material/Diffuse.cu`). The shader has two closest-hit programs, one for radiance rays and one for photons. Headers are omitted.

```
using namespace optix;

rtDeclareVariable(uint2, launchIndex, rtLaunchIndex, );
rtDeclareVariable(RadiancePRD, radiancePrd, rtPayload, );
rtDeclareVariable(PhotonPRD, photonPrd, rtPayload, );
rtDeclareVariable(optix::Ray, ray, rtCurrentRay, );
rtDeclareVariable(float, tHit, rtIntersectionDistance, );

rtDeclareVariable(float3, geometricNormal, attribute geometricNormal, );
rtDeclareVariable(float3, shadingNormal, attribute shadingNormal, );

rtBuffer<Photon, 1> photons;
rtBuffer<Hitpoint, 2> raytracePassOutputBuffer;
rtDeclareVariable(rtObject, sceneRootObject, , );
rtDeclareVariable(uint, maxPhotonDepositsPerEmitted, , );
rtDeclareVariable(float3, Kd, , );

RT_PROGRAM void closestHitRadiance()
{
    float3 worldShadingNormal = normalize( rtTransformNormal(
        RT_OBJECT_TO_WORLD, shadingNormal ) );
    float3 hitPoint = ray.origin + tHit*ray.direction;

    radiancePrd.flags |= PRD_HIT_NON_SPECULAR;
    radiancePrd.attenuation *= Kd;
    radiancePrd.normal = worldShadingNormal;
    radiancePrd.position = hitPoint;
    radiancePrd.lastTHit = tHit;
    if(radiancePrd.flags & PRD_PATH_TRACING)
    {
        radiancePrd.randomNewDirection = sampleUnitHemisphereCos(
            worldShadingNormal, getRandomUniformFloat2(&radiancePrd.
            randomState));
    }
}
```

```
RT_PROGRAM void closestHitPhoton()
{
    float3 worldShadingNormal = normalize( rtTransformNormal(
        RT_OBJECT_TO_WORLD, shadingNormal ) );
    float3 hitPoint = ray.origin + tHit*ray.direction;
    float3 newPhotonDirection;

    if(photonPrd.depth >= 1 && photonPrd.numStoredPhotons <
        maxPhotonDepositsPerEmitted)
    {
        Photon photon (photonPrd.power, hitPoint, ray.direction,
            worldShadingNormal);
        photons[photonPrd.pm_index + photonPrd.numStoredPhotons] = photon;
        photonPrd.numStoredPhotons++;
    }

    photonPrd.power *= Kd;
    photonPrd.weight *= fmaxf(Kd);

    // Use russian roulette sampling to limit the length of the path

    if( photonPrd.depth >= PHOTON_TRACING_RR_START_DEPTH)
    {
        float probContinue = favgf(Kd);
        float probSample = getRandomUniformFloat(&photonPrd.randomState);
        if(probSample >= probContinue )
        {
            return;
        }
        photonPrd.power /= probContinue;
    }

    photonPrd.depth++;
    if(photonPrd.depth >= MAX_PHOTON_TRACE_DEPTH || photonPrd.weight < 0.001)
    {
        return;
    }

    if(photonPrd.numStoredPhotons >= maxPhotonDepositsPerEmitted)
    {
        return;
    }

    newPhotonDirection = sampleUnitHemisphereCos(worldShadingNormal,
        getRandomUniformFloat2(&photonPrd.randomState));
    optix::Ray newRay( hitPoint, newPhotonDirection, RayType::PHOTON, 0.0001
        );
    rtTrace(sceneRootObject, newRay, photonPrd);
}
```

C

PARTICIPATING MEDIUM SHADER

We also include the participating medium closest-hit programs for radiance and photons (`material/ParticipatingMedium.cu`). Headers are omitted.

```
using namespace optix;

rtDeclareVariable(rtObject, volumetricPhotonsRoot, , );
rtDeclareVariable(rtObject, sceneRootObject, , );
rtDeclareVariable(uint2, launchIndex, rtLaunchIndex, );
rtDeclareVariable(RadiancePRD, radiancePrd, rtPayload, );
rtDeclareVariable(VolumetricRadiancePRD, volRadiancePrd, rtPayload, );
rtDeclareVariable(ShadowPRD, shadowPrd, rtPayload, );
rtDeclareVariable(PhotonPRD, photonPrd, rtPayload, );
rtDeclareVariable(TransmissionPRD, transmissionPrd, rtPayload, );
rtDeclareVariable(optix::Ray, ray, rtCurrentRay, );
rtDeclareVariable(float, tHit, rtIntersectionDistance, );

rtDeclareVariable(float3, geometricNormal, attribute geometricNormal, );
rtDeclareVariable(float3, shadingNormal, attribute shadingNormal, );

rtBuffer<Photon, 1> photons;
rtBuffer<Photon, 1> volumetricPhotons;
rtDeclareVariable(float, sigma_a, , );
rtDeclareVariable(float, sigma_s, , );
rtDeclareVariable(float3, Ks, , );
rtDeclareVariable(float3, Kd, , );
rtDeclareVariable(float, indexOfRefraction, , );
rtDeclareVariable(uint, maxPhotonDepositsPerEmitted, , );

RT_PROGRAM void closestHitRadiance()
{
    const float sigma_t = sigma_a + sigma_s;
    float3 worldShadingNormal = normalize(rtTransformNormal(
        RT_OBJECT_TO_WORLD, shadingNormal));
    float3 hitPoint = ray.origin + tHit*ray.direction;
    bool isHitFromOutside = hitFromOutside(ray.direction, worldShadingNormal)
        ;
    double tHitStack = tHit;

    if(isHitFromOutside)
    {
```

```
        float3 attenSaved = radiancePrd.attenuation;

        // Send ray through the medium
        Ray newRay(hitPoint, ray.direction, RayType::
            RADIANCE_IN_PARTICIPATING_MEDIUM, 0.01);
        rtTrace(sceneRootObject, newRay, radiancePrd);

        float distance = radiancePrd.lastTHit;
        float transmittance = exp(-distance*sigma_t);

        VolumetricRadiancePRD volRadiancePrd;
        volRadiancePrd.radiance = make_float3(0);
        volRadiancePrd.numHits = 0;
        volRadiancePrd.sigma_t = sigma_t;
        volRadiancePrd.sigma_s = sigma_s;

        // Get volumetric radiance

        Ray ray(hitPoint, ray.direction, RayType::VOLUMETRIC_RADIANCE,
            0.0000001, distance);
        rtTrace(volumetricPhotonsRoot, ray, volRadiancePrd);

        /* Multiply existing volumetric transmittance with current
            transmittance, and add gathered volumetric radiance
         from this path */

        radiancePrd.volumetricRadiance *= transmittance;
        radiancePrd.volumetricRadiance += attenSaved*volRadiancePrd.radiance;
        radiancePrd.attenuation *= transmittance;
    }
    else
    {
        /* We are escaping the boundary of the participating medium, so we'll
             compute the attenuation and volumetric radiance for the
            remaining path
         and deliver it to a parent stack frame. */
        Ray newRay = Ray(hitPoint, ray.direction, RayType::RADIANCE, 0.01);
        rtTrace(sceneRootObject, newRay, radiancePrd);
    }

    radiancePrd.lastTHit = tHitStack;
}

/*
//
*/

RT_PROGRAM void closestHitPhoton()
{
    const float sigma_t = sigma_a + sigma_s;
```

```
photonPrd.depth++;
float3 worldShadingNormal = normalize( rtTransformNormal(
    RT_OBJECT_TO_WORLD, shadingNormal ) );
float3 hitPoint = ray.origin + tHit*ray.direction;
bool hitInside = (dot(worldShadingNormal, ray.direction) > 0);

// If we hit from the inside with a PHOTON_IN_PARTICIPATING_MEDIUM ray,
    we have escaped the boundry of the medium.
// We move the ray just a tad to the outside and continue ray tracing
    there
if(hitInside && ray.ray_type == RayType::PHOTON_IN_PARTICIPATING_MEDIUM)
{
    Ray newRay = Ray(hitPoint+0.0001*ray.direction, ray.direction,
        RayType::PHOTON, 0.001, RT_DEFAULT_MAX);
    rtTrace(sceneRootObject, newRay, photonPrd);
    return;
}

float sample = getRandomUniformFloat(&photonPrd.randomState);
float scatterLocationT = - logf(1-sample)/sigma_t;
float3 scatterPosition = hitPoint + scatterLocationT*ray.direction;
int depth = photonPrd.depth;

/* We need to see if anything obstructs the ray in the interval from the
    hitpoint to the scatter location.
   If nothings obstructs then we scatter at eventPosition. Otherwise, the
       photon continues on its path and we don't do anything
   when we return to this stack frame. We keep the photonPRD depth on the
       stack to compare it when the rtTrace returns. */

Ray newRay(hitPoint, ray.direction, RayType::
    PHOTON_IN_PARTICIPATING_MEDIUM, 0.001, scatterLocationT);
rtTrace(sceneRootObject, newRay, photonPrd);

/* If depth is unmodified, no surface was hit from hitpoint to
    scatterLocation, so we store it as a scatter event.
   We also scatter a photon in a new direction sampled by the phase
      function at this location. */

if(depth == photonPrd.depth)
{
    const float scatterAlbedo = sigma_s/sigma_t;

    if(getRandomUniformFloat(&photonPrd.randomState) >= scatterAlbedo)
    {
        return;
    }

    // Store photon at scatter location
```

```
        int volumetricPhotonIdx = photonPrd.pm_index % NUM_VOLUMETRIC_PHOTONS
            ;
        volumetricPhotons[volumetricPhotonIdx].power = photonPrd.power;
        volumetricPhotons[volumetricPhotonIdx].position = scatterPosition;
        atomicAdd(&volumetricPhotons[volumetricPhotonIdx].numDeposits, 1);

        // Check if we have gone above max number of photons or stack depth
        if(photonPrd.depth >= MAX_PHOTON_TRACE_DEPTH)
        {
            return;
        }

        // Create the scattered ray with a direction given by importance
            sampling of the phase function
        float3 scatterDirection = sampleUnitSphere(getRandomUniformFloat2(&
            photonPrd.randomState));
        Ray scatteredRay(scatterPosition, scatterDirection, RayType::PHOTON,
            0.001, RT_DEFAULT_MAX);
        rtTrace(sceneRootObject, scatteredRay, photonPrd);
    }
}
```