THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Real-Time GPU Techniques for Advanced Lighting Phenomena

Markus Billeter

Division of Computer Engineering
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2014

Real-Time GPU Techniques for Advanced Lighting Phenomena

Markus Billeter
Göteborg, 2014
ISBN 978-91-7385-993-6

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 3674
ISSN 0346-718X

Technical Report 110D
Department of Computer Science and Engineering
Computer Graphics Research Group

Division of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000

Contact information:
Markus Billeter
Division of Computer Engineering
Dept. of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden

Phone: +46 (0)31 772 52 12
Fax: +46 (0)31 772 36 63
Email: markus@newq.net
billeter@chalmers.se
URL: http://link.newq.net/home
http://www.cse.chalmers.se/~billeter

(Scan to download PDF.)

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2014

Real-Time GPU Techniques for Advanced Lighting Phenomena

Markus Billeter
Division of Computer Engineering, Chalmers University of Technology

Abstract

In the real world, the visual perception of an object is completely determined by the object's interactions with light. One large application of computer graphics is to visualize virtual objects and worlds in a fashion that is familiar to humans. Successfully emulating light and its effects on virtual objects therefore plays a central role.

The papers included in this thesis mainly explore improved methods of computing the effects of light in various settings. The focus is on doing so in real-time for interactive applications. Two papers target capturing the visual effects of light traveling through a participating medium (a medium such as fog or smog). The first of these papers presents a method that can be used to render shafts of light/volumetric shadows in real time. The second paper extends this to include additional effects associated with participating media, including, for example, indirect illumination of surfaces from light scattered in the medium.

Next, two papers explore real-time rendering with many light sources. One paper presents a method to efficiently render in the presence of and manage thousands of light sources and demonstrates scaling up to one million lights. The other paper focuses on rendering on mobile devices (such as smartphones and tablet devices), and investigates the possibility of off-loading rendering tasks to a remote server. The paper presents one approach where a server computes indirect illumination represented by virtual light sources. The client retrieves these virtual light sources from the server and uses an adapted version of the previously presented many-lights technique for rendering.

Graphics processing units (GPUs) play a central role in all these techniques. Thus, the first paper included in this thesis discusses efficient implementations of fundamental building blocks for programming GPUs. In particular, it presents an efficient implementation of the *stream compaction* operation. It further discusses the programming strategy that makes the implementation efficient and demonstrates several related fundamental operations developed using that strategy.

Keywords: Computer Graphics, GPGPU, Rendering, Shading, Participating Media, Scattering, Volumetric Shadows, Many-Light, Mobile Graphics.

Acknowledgments

The first round of thanks goes, of course, to my supervisor Ulf. He's given an incredible amount of advice, support and feedback over the years ... And a huge thanks to him for the opportunity to work here in the Graphics Research Group. That group would not be quite what it is without the other people in it: Erik, Ola and Viktor. Working with you has been a blast! There are many more people at Chalmers that have made my time here enjoyable. A thank you to all of these people, and especially to Per Stenström, my examiner, and Morten Fjeld, my co-supervisor.

A special thank you to Lei Yang and Liu Ren for giving me the opportunity to visit and work in the Visual Computing Group at Bosch Research. My time in California was a great experience. I doubt I've visited as many places in such a short period of time ever before or since - here's a shoutout to all my co-travelers that made those trips a reality. (Though, next time I go to Hawaii, I want to spend more than 4 days there.)

A special mention goes to a group of people that I met in 2003 during my first days at Chalmers; a group whom I now count to my close friends. Unfortunately, we've gone from meeting daily, to weekly lunches (apparently mostly on days other than Thursday), to ... less regularly, as people have been moving away and spreading out. (Spreading out across the whole world, I could say, although one of us single-handedly accounts for most of the spread by making it all the way to Australia.)

Finally, thanks to my friends, my parents and my sister. All of you have had to listen to my weird ideas that (sometimes) seemingly barely make sense, and have mostly done so quite graciously.

Because this thesis is, in essence, about enabling interactivity, I want to make the thesis itself a tiny little interactive. So, if you want, you can write your name in the following blank spot: A special thank you to _____ for reading my ramblings so far. (And don't give up reading just quite yet – there's more to come.)

Credits

This document uses icons from the "Silk Icons Set 1.3" by Mark James. The icons are licensed under the Creative Commons Attribution 2.5 License, and can be found at http://www.famfamfam.com/lab/icons/silk/.

List of Appended Papers

This thesis is a summary of the following papers. References to the papers will be made using roman numerals.

Paper I – **Markus Billeter**, Ola Olsson, Ulf Assarsson,
Efficient Stream Compaction on Wide SIMD Many-Core Architectures,
HPG '09: Proc. of the Conf. on High Performance Graphics (pp 159–166)
New Orleans, Louisiana, 2009

Paper II – **Markus Billeter**, Erik Sintorn, Ulf Assarsson,
Real Time Volumetric Shadows using Polygonal Light Volumes,
HPG '10: Proc. of the Conf. on High Performance Graphics (pp 39–45)
Saarbrucken, Germany, 2010

Paper III – **Markus Billeter**, Erik Sintorn, Ulf Assarsson,
Real Time Multiple Scattering using Light Propagation Volumes,
I3D '12: Interactive 3D Graphics and Games (pp 119–126)
Costa Mesa, CA, USA, 2012

Paper IV – Ola Olsson, **Markus Billeter**, Ulf Assarsson,
Clustered Deferred and Forward Shading,
HPG '12: Proc. of the Conf. on High Performance Graphics (pp 87–96)
Paris, France, 2012

Paper V – **Markus Billeter**, Lei Yang, Liu Ren, Ulf Assarsson,
Cloud-Assisted Indirect Illumination on Mobile Devices,
manuscript - under revision

Other Contributions

The following contributions by the same author are not directly included in this thesis.

$\mathcal{A}$ – Multi-Light Rendering On Mobile Devices,
Markus Billeter, Ola Olsson, Ulf Assarsson,
(abstract/talk - under submission)

$\mathcal{B}$ – Implementing Efficient Virtual Shadow Maps from Many Lights,
Ola Olsson, Erik Sintorn, Viktor Kämpe, **Markus Billeter**, Ulf Assarsson,
(abstract/talk - under submission)

$\mathcal{C}$ – Encoding Binary Voxel Grids for Free Viewpoint Video,
Viktor Kämpe, **Markus Billeter**, Erik Sintorn, Ulf Assarsson,
(under revision)

$\mathcal{D}$ – Efficient Virtual Shadow Maps for Many Lights,
Ola Olsson, Erik Sintorn, Viktor Kämpe, **Markus Billeter**, Ulf Assarsson,
I3D '14: Interactive 3D Graphics and Games, San Francisco, CA, USA, 2014
10.1145/2556700.2556701
http://youtu.be/jjAE0h5VNT0
http://link.newq.net/thesis/VirtualShadowMaps

$\mathcal{E}$ – Two-Level Grids for Ray Tracing on GPUs,
Javor Kalojanov, **Markus Billeter**, Phillipp Slusallek,
EG '11: Eurographics, Llandudno, UK, 2011
10.1111/j.1467-8659.2011.01862.x
http://link.newq.net/thesis/TwoLevelGrids

$\mathcal{F}$ – Tiled Forward Shading,
Markus Billeter, Ola Olsson, Ulf Assarsson,
GPU Pro 4 – Advanced Rendering Techniques, A K Peters / CRC Press, 2013
http://www.crcpress.com/product/isbn/9781466567436

$\mathcal{G}$ – Tiled and Clustered Forward Shading,
Ola Olsson, **Markus Billeter**, Ulf Assarsson,
SIGGRAPH '12: SIGGRAPH Talks, Los Angeles, CA, USA, 2012
10.1145/1572769.1572795
http://link.newq.net/thesis/ClusteredForward

Table of Contents

Part I
Summary

1 Introduction

Computer graphics has, since its inception a little bit over half a century ago, seen a tremendous growth. Nowadays, computer graphics permeates modern society: we find applications everywhere, ranging from entertainment to research, design and even medicine. Because of this wide range of applications, the subject of computer graphics itself now encompasses a vast number of sub-topics. The work in this thesis focuses on one such sub-topic: *real-time rendering*.

Sections 1.1 and 1.2 introduce the subject of real-time rendering and some of the most important concepts that the remainder of the thesis relies on. Readers familiar with this field can skip (or just quickly skim) the sections leading up to Section 1.3 Overall Objective and Problem Statement.

1.1 Real-time Rendering

This thesis primarily deals with the topic of real-time rendering. In computer graphics, *rendering* refers to the process of generating a 2D image from a model. This generated image is then, for example, shown to the user on a screen; however, this needs not be the case - the generated image could just as well be used as an input for some other computations.

To view a model that changes over time, we display new images rapidly, where each image depicts the model at a fixed point in time. If we manage to display the images rapidly enough, we create the illusion of a smooth *animation*. This typically requires us to display in excess of 20 images per second. If we fail to do so, the animation will look jerky and bad.

A rate of 20 images (*frames*) per second is a lower limit. While a typical movie runs at 24 or more frames per second, the recent HFR (*h*igh *f*rame *r*ate) movies double that. Players of action games might expect at least 60 frames per second (*FPS*) and with stereo rendering thrown into the mix, in excess of 120 FPS might be necessary.

We further distinguish between *real-time* and *off-line* rendering. Real-time rendering is required when we wish to be able to *interact* with our virtual world. Our interactions change the world (and thus the model that we need to render), which prevents pre-computation of images. Games typically require real-time rendering techniques because of this - there is no way to know what decisions the player makes at any given point in the game.

Given the need to display at least 20 images per second, we have less than 50ms to render and display each frame. That represents a rather comfortable upper bound: at 120 FPS, our estimated 50ms frame-budget is decimated to around 8ms. This is further compounded by the fact that it is not always possible to spend **all** of this time on rendering.

In contrast, off-line rendering relaxes the restrictions on rendering speed. For instance, we cannot interact with a movie and affect changes in it. This allows

the producers of the movie to pre-compute all images. When pre-computing images off-line, the rendering process is allowed to take from many seconds to hours and possibly even days[1].

Techniques for movies and other off-line rendering applications nowadays focus more on quality than raw performance. In the extreme end of this balance, we find *predictive* rendering, where the goal is to model reality so accurately that the rendered image can be used to predict the look of objects in real-life. This helps designers, for instance, in determining whether their products will look as expected under various real-life conditions.

Real-time rendering represents in many ways an opposite to predictive rendering. In real-time rendering, performance is paramount. Not only must techniques support real-time performance, but they should also have predictable worst-cases so that smooth animation can be ensured. More often than not, this requires trading visual quality and complexity for speed. One large aspect of real-time rendering research is finding appropriate models that display certain desired features but remain simple enough to enable evaluation in real-time.

One example of this is found in the participating media part of this thesis. A participating medium is a volumetric effect where many small particles affect light passing through the volume. Examples of participating media include fog, clouds and smoke (see Figure 2). The physics of most participating media are well known, and numerical methods can be used to accurately solve light transport through such media, at least on limited scales. However, rendering applications often display scenes on macroscopic scales, and suddenly the accurate simulations become intractable (even in off-line settings). Section 3 presents participating media from a real-time rendering perspective, and discusses models describing participating media that enable inclusion of some visually important effects attributed to such media.

Another aspect of real-time rendering is the use of novel and clever algorithms that reduce the amount of work needed to achieve a certain result, or that compute the result more efficiently. The many-lights part of the thesis (Section 4) details such an improvement.

A third aspect that contributes to the ever-improving visuals possible in real-time rendering are advances on the hardware side. The hardware dedicated to real-time rendering, the *graphics processing unit* (GPU), has evolved at a stunning speed in the past few years. GPUs play a large role in this thesis: a common theme in all presented methods is that they attempt to utilize the GPU (more) efficiently. The next section introduces the GPU and other tools that are used to develop the methods.

[1] Of course, off-line rendering people care a lot about performance, too. After all, a movie needs to be finished at some point. Rendering time is not free either - at the very least it consumes electricity. And artists producing the movie might not be able to continue work without being able to inspect their results.

(a) *(b)*

Figure 1. Left: NVIDIA GTX Titan GPU (kindly donated to our research group by the NVIDIA corporation). The GTX Titan is capable of a theoretical throughput of 4.5 TFLOPS and 288 GBytes/sec. Right: Samsung Galaxy Note 10.1 powered by an ARM Mali-T628 Mobile GPU, rated at 109 GFLOPS and 15 GBytes/sec. The photo shows the tablet running the client program described in Paper **V**.

1.2 Tools of Trade

The term GPU was introduced relatively recently (1999) by NVIDIA when unveiling the GeForce 256. The GeForce 256 is of course by far not the first type of hardware dedicated to computer graphics - many types of graphics accelerators existed before it. The GeForce 256 was the first to offload parts of the geometry processing from the CPU. Nevertheless, the GeForce 256 remains solely a graphics accelerator.

Accelerating graphics is still one of the main reasons why GPUs exist, but a modern GPU is no longer just a graphics accelerator. GPUs have become tremendously powerful: the GPU shown in Figure 1a is capable of sustaining a theoretical throughput of 4.5 TFLOPS and a memory bandwidth of 288 GBytes/sec. This has sparked an increased interest in using GPUs for other purposes than graphics - for more general computations. The term GPGPU, a combination of the phrase 'general purpose' and GPU, refers to this trend.

Because of this, GPUs exhibit a sort of duality. One one hand, GPUs are devices dedicated to accelerating rendering. On the other hand, they can run general purpose algorithms. As programmers, we see this duality in how we access the GPU. When targeting graphics problems, *application programming interfaces* (API) like OpenGL and Direct3D are used. For programming of general algorithms, we instead use APIs and tools like CUDA or OpenCL[2]. Really interesting things happen when these worlds are combined to leverage both the dedicated graphics features of GPUs and the flexibility of the GPU's general purpose aspects. Many of the methods presented in this thesis use a combination of both OpenGL and CUDA.

[2]Or, more recently, C++AMP and various types of compute shaders.

(a) (b) (c)

Figure 2. Participating media in the real world. Images (a) and (b) display scattering in relatively homogeneous fog. The photos were taken on Gibraltargatan, just outside of Chalmers. Image (c) shows a low cloud moving past the cathedral of Fribourg (Switzerland). The cathedral is illuminated by several strong light sources, the light of which is scattered in the cloud, making the cloud visible. The density of the cloud varies; this is an example of a non-homogeneous participating medium.

Mobile GPUs have likewise become powerful in their own right. Figure 1b displays a Samsung tablet device powered by an ARM Mali-T628 mobile GPU. This GPU is rated at 109 GFLOPS and has approximately 15 GBytes/sec of memory bandwidth. For mobile devices, we rely on OpenGL|ES 2.0 and more recently OpenGL|ES 3.0. These are APIs based on the desktop OpenGL, but with a somewhat reduced feature set suitable for mobile devices. GPGPU computing on mobile GPUs is also starting to appear. However, so far only very few combinations of mobile GPUs, devices and operating systems provide official support for e.g. OpenCL.

1.3 Overall Objective and Problem Statement

The overarching goal of my work is to develop improved methods for real-time lighting. The methods should be deployable on modern (high-end) consumer-grade hardware; I therefore focus on algorithms that run on modern GPUs.

In recent years, GPUs have undergone significant changes in which they have transformed from devices with mainly fixed-function capabilities targeting computer graphics to devices capable of running general purpose programs. One open question is how these new capabilities can be harnessed to produce better algorithms. Yet, GPUs retain many capabilities directly related to rendering. We should not ignore these capabilities, but take advantage of them when we develop new methods. By taking full advantage of the GPU, i.e., by taking advantage of both the general-purpose aspects and fixed functionality, we can produce visually more convincing effects while retaining the real-time performance that interactive applications require.

I examine three sub-topics in this thesis. First, I focus on efficient general-purpose building-blocks in the form of GPU parallel primitives. Secondly, I explore rendering in participating media. Finally, I investigate improvements to many-light rendering methods.

Parallel Primitives. To effectively develop efficient parallel algorithms, we need fundamental building blocks that utilize GPUs well. In Paper **I**, I focus on stream compaction, which is one of these building blocks. I familiarize myself with the GPU and how it is programmable with CUDA. This results in an efficient implementation of stream compaction and several related operations. Additionally, Paper **I** examines a general model and strategy that allows reasoning about and assists development of efficient GPU algorithms.

Participating Media. Rendering often only takes the geometry of the scene into account. Yet, including participating media can greatly improve perceived visual quality. Effects such as shafts of light present the viewer with important visual cues, like depth and positions of light sources. Additionally, presence of participating media can drastically change the mood of a scene. Figure 2 displays several real-world photos where participating media play an important role.

First, I investigate single scattering in Paper **II**, in order to enable efficient rendering of volumetric shadows (also known as shafts of light).

Secondly, I look at multiple scattering in Paper **III**. The goal is to extend rendering with participating media to include several additional visually important effects, including indirect illumination of otherwise shadowed surfaces, and scattering in shadowed regions of the medium.

Many-Light Shading. Most real-world scenes contain many light sources (see, e.g., Figure 3a). In addition to these primary light sources, secondary virtual light sources can be used to emulate indirect lighting effects (Figure 3b). Scenes can therefore end up containing a large number of light sources that must be considered during rendering – for example, Ferrier and Coffin [2011] mention game scenes with several thousands of light sources.

In Paper **IV**, I explore rendering and management of large numbers of light sources. The aim is to avoid fundamental problems with previous approaches and to produce a more probust method.

Finally, in Paper **V**, I look into rendering on mobile devices such as tablets and smartphones. The goal is to develop methods that bring advanced and modern lighting to these devices. The methods should utilize the rapidly improving mobile GPUs (and CPUs), but also take advantage of the always-connected nature of modern mobile devices. The chosen approach uses a many-light rendering method derived from the technique developed in Paper **IV** on the client and uses remote servers to assist computing and managing indirect illumination.

1.4 Main Contributions

The contributions presented in this thesis are spread across the three sub-topics of general parallel algorithms, rendering with participating media and rendering with many lights. However, a common denominator of all presented techniques in this thesis is that they rely on being able to efficiently utilize the GPU.

(a) (b) (c)

Figure 3. Scenarios for many-light rendering. Settings with many lights occur in the real world (a), as seen in this photo of Las Vegas by night. Many-light–techniques can be used to emulate indirect lighting (b). The indirect light from the red curtain, the red 'reflection' on the floor, is produced using six (invisible) directional light sources placed on the curtain. The positions of these six lights are visualized in the rightmost image (c) using magenta spheres. This image also displays shading resulting from only the primary light. (Photo (a) by Daniel Lutz, 2012. Used with permission.)

Paper **I** explores parallel primitives on GPUs. It presents an efficient stream compaction implementation. It discusses a model and strategy that allows development of better parallel algorithms. This model and strategy is used to implement additional parallel operations. Paper **I** also builds a foundation for the future techniques.

Paper **II** and Paper **III** explore rendering in participating media. Paper **II** focuses on single scattering. It presents a GPU-friendly method to produce volumetric shadows. The follow-up work, Paper **III**, extends this to consider other forms of scattering. It presents a real-time method that enables rendering with a number of new effects when compared to Paper **III**, including:

- Indirect illumination of surfaces from scattered light.
- Indirect illumination by light reflected of surfaces.
- Scattering in parts of the medium that are not directly illuminated.
- Scattering of light reflected from surfaces.

Paper **IV** investigates rendering with many lights. It presents an improved method for rendering in many-light settings that significantly reduces the number of lighting computations. The method performs more robustly with respect to varying views (see, e.g., in-game screenshots presented by Persson and Olsson [2013]). The method is demonstrated to scale reasonably up to at least 1M light sources.

Paper **V** leverages the work from Paper **IV** to create an adapted many-light

rendering method that works on current mobile devices. The adapted method is used for rendering on the mobile client, which receives information about lights from a remote server. The received lights represent e.g., indirect illumination that the server can efficiently compute (and share among many clients). Paper **V** shows that the technique presented in Paper **IV** can be scaled to work on current mobile devices. It explores what kind of computations can be off-loaded to a remote server and shows that indirect illumination is one possibility. Indirect illumination is further client-independent, and a solution for indirect illumination can be shared among many clients, making it an excellent candidate for off-loading to a central sever.

1.5 Structure of the Thesis

This thesis is based on five papers. The topic of the first paper, GPU parallel primitives, is discussed in Section 2 and revolves around general purpose parallel algorithms. Section 2.1 shortly summarizes Paper **I**. Paper **II** and Paper **III** consider participating media, which is the topic of Section 3. The papers are further summarized in Sections 3.1 and 3.2, respectively. The third topic of many-light rendering is treated in Section 4. Sections 4.1 and 4.2 summarize Paper **IV** and Paper **V**, the two papers on this topic.

Section 5 further discusses the methods and techniques presented in the papers. Section 5.1 presents recent advances and improvements to the clustered shading method introduced in Paper **IV**. Section 5.2 documents some of the experiences from implementing tiled shading on mobile devices, gathered while developing Paper **V**. It also details the reasons for ultimately switching to clustered shading for our work in Paper **V**.

2 Parallel Primitives

A core objective of this work is to present algorithms and methods that efficiently utilize the GPU. Paper **I** focuses on this aspect and presents an algorithm for stream compaction. The presented strategy is, however, valid for related operations, such as reductions, scans and stream splits.

These operations serve as building blocks for more advanced procedures. For instance, sorting functions can be built from consecutive stream compactions and/or stream splits. Sorting itself is also a fundamental building block for many methods. For example, we use sorting and (segmented) reductions to construct a bounding volume hierarchy in Paper **IV**.

Sequential variants of these algorithms are often quite simple. A reduction operation *reduces* multiple elements to a single value using some operator. A reduction with addition as its operator computes from N input values a_i

$$r = \sum_{i=0}^{N} a_i.$$

A scan performs a reduction for each input element with all preceding elements as the reduction's input. Using addition as the reduction's operator, it computes the following:

$$r_i = \sum_{j=0}^{i-1} a_j. \tag{1}$$

This special variant of the scan is also known as a *prefix sum*. The prefix sum (and the scan operation in general) comes in two flavours: inclusive and exclusive. Equation (1) describes the former, as the i:th input is excluded from the computation of the i:th output. The inclusive flavour instead includes this last element.

Conceptually, stream compaction filters elements from the input. A predicate determines whether each input element is considered valid and should be kept, or if it is invalid and ought to be discarded. Stream compaction places the valid elements in an compact output buffer. Listing 1 shows a sequential pseudo-code. Stream split is a closely related operation: while stream compaction discards the invalid elements, stream split places invalid elements in the second part of the output buffer, i.e., after the last valid element.

```
1 j = 0;
2 for( i = 0; i < N; ++i )
3 {
4     if( a[i] valid )
5     {
6         r[j] = a[i];
7         ++j;
8     }
9 }
```

Listing 1. Sequential stream compaction. Input elements `a[i]` are copied to the output buffer `r[i]` if they are considered valid. Validity is determined using a predicate function; for example, the predicate could consider non-zero elements to be valid.

Implementing a sequential stream compaction is trivial, as demonstrated by Listing 1. The scan and reduction operations are even simpler. However, the parallel versions require a bit more thought. For stream compaction, we run into the problem that the output location of each elements depends on the state of *all* previous elements. The problem is not trivially data-parallel.

Blelloch [1990] presents a strategy that uses an exclusive prefix sum to implement parallel stream compaction. Many modern implementations of stream compaction still rely on this strategy (including the method presented in Paper I). Blelloch [1990] also discusses implementations of parallel scans and reductions, as does e.g., Hillis and Steele [1986]. However, both target somewhat outdated hardware, and we would like to use the parallel primitives on modern GPUs.

2.1 Paper I

Problem. Many parallel methods and algorithms perform much better and are much easier to realize when their input is stored in a compact range. However, the result of many parallel methods is sparse. This is where the stream compaction comes in: it compacts the intermediate results before these are input into the next parallel processing step.

For instance, extracting clusters in the page-table–based clustering method of Paper **IV** represents one example application of stream compaction. The sorting-based clustering relies on high-performance radix-sort, which can be built from consecutive stream splits[3]. The light hierarchy of Paper **IV** is built using sorting and (segmented) reductions.

We need efficient implementations of stream compaction and related operations like reductions, scans and stream splits. Parallel implementations of these operations have been studied extensively (see e.g., the work by Hillis and Steele [1986] and Blelloch [1990]), and several GPU implementations exist. However, many of these implementations are designed for older GPUs that, for example, lack support for random write access to memory [Horn, 2005; Roger et al., 2007; Sengupta et al., 2006]. We would also like to avoid the need for temporary storage as much as possible. For instance, we would like to avoid storing the input to and the results of an explicit prefix sum (unlike e.g., the CUDA implementation suggested by Harris et al. [2008]).

We also found very early that our own implementation of stream compaction would outperform other publicly available implementations, such as, for example, the ones included in the CUDPP library (one of the early libraries to implement parallel primitives in CUDA).

Methodology. We developed an efficient parallel stream compaction algorithm using CUDA. Our algorithm avoids over-parallelizing the problem by dividing the input into compact chunks such that the majority of work can be performed serially and independently by CUDA warps. The implementation targets the then-current NVIDIA GTX280 GPU. We evaluated performance of our implementation and compared it to other implementations. We additionally implemented the scan operation, the stream split and a radix sort based on the stream split. We compared performance of our implementations to competing methods.

Algorithm Overview. We consider a CUDA warp to be an independent virtual processor with a SIMD width of $S = 32$. Threads of a warp execute in lockstep, and therefore no synchronization is necessary between the threads of a single warp. The number of virtual processors P must be sufficiently large to enable latency hiding on the GPU.

We assume that the number of input elements N is larger than $P \times S$. The input

[3]The radix sort is performed on many 1024-element (32^2) groups of samples, rather than globally across all samples in the frame buffer.

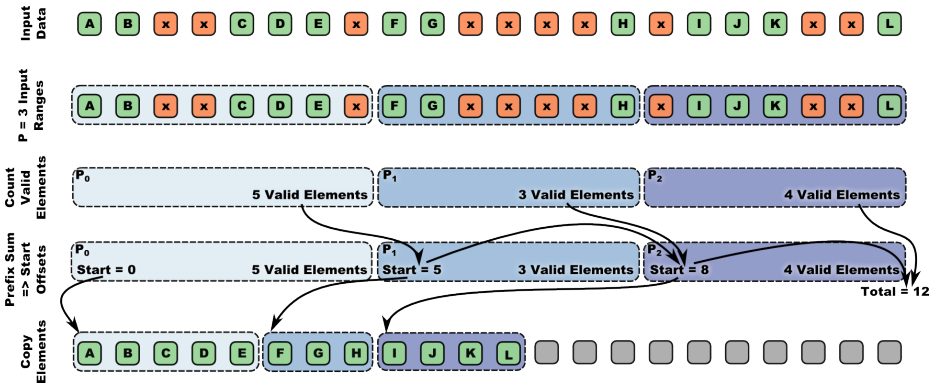


Figure 4. Illustration of the compaction algorithm. Elements marked with 'x' are invalid and to be removed, while elements 'A' through 'L' should be kept. Our compaction algorithm is stable, and therefore the order of the valid elements remains unchanged. In this example we use $P = 3$ processors with $N = 22$ input elements. Processor 1 gets one extra element. The SIMD-nature of the processors is not visible at this level of detail.

is first subdivided into P compact ranges, and each range is assigned to a virtual processor. Compaction then occurs in three steps (also illustrated in Figure 4):

1. Each processor independently counts the number of valid elements in its input range.
2. A single CUDA block performs a scan of the P output elements of the previous step.
3. Each processor independently copies valid elements from its input range to its output range, as identified by the offsets computed in Step 2.

Most of the work occurs in Steps 1 and 3. In these two steps, each warp can process its input elements independently from all other warps.

Contributions. We presented a fast parallel stream compaction method, which outperformed other known implementations by a factor of $3\times$. We only use in the order of $O(P)$ elements of temporary storage. Our implementation is stable, i.e., the relative order of elements remains unchanged, which is essential when implementing e.g., a radix sort based on compaction.

We describe a programming model and strategy for attacking problems that are not completely data parallel (like stream compaction). Our strategy incorporates knowledge of the underlying hardware and thereby allows better performing implementations of parallel operations. With this strategy, we implement other parallel operations, including scans and parallel sorting. The performance of these operations compares favorably to other concurrent implementations.

Our CUDA-implementations of these parallel operations were made freely

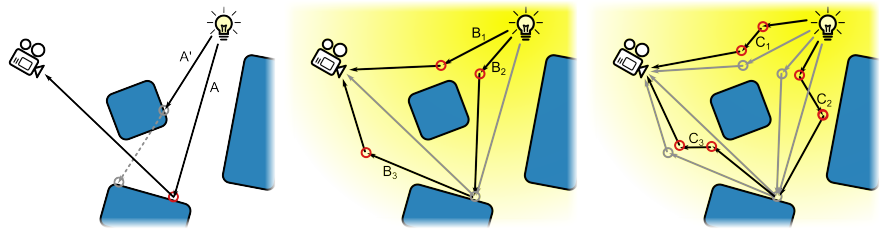


Figure 5. Different types of paths that light can travel on from a light source to the camera. Traditionally, real-time rendering methods consider paths of type A that connect the light source to the camera via a surface (a). In a participating medium, new paths become possible. Paths of type B_i contain one single bounce in the medium (b) and fall into the category of single scattering. Paths of type C_i contain two or more bounces in the medium (multiple scattering). Figure (c) illustrates two bounces only, but we include paths with a higher number of bounces in the class of paths of type C_i.

available as a small header-only library released under the MIT license.

3 Participating Media

Real-time rendering often focuses on light paths of type A (Figure 5a), i.e., paths where light travels from a light source to a surface, from where it is reflected towards the eye. In fact, just identifying paths of type A', where the path from the light source to the surface is blocked, remains problematic (some very recent publications that attack this problem include Dou et al. [2014]; Olsson et al. [2014] and Sintorn et al. [2014]).

In the presence of a participating medium, several additional paths become possible and must be considered. We distinguish between two classes of paths. Paths of type B_i (Figure 5b) fall into the category of single scattering. Here, paths contain one bounce in the medium. Many techniques, including the one presented in Paper **II**, further only consider paths of type B_1, where light travels from the light source into the medium, is scattered once, and then reaches the camera.

Paths of type C_i (Figure 5c) are scattered several times by the medium. Technically, paths of type B_i are a subset of type C_i, but in Paper **III**, where we consider paths of type C_i, it is convenient to separate these two types.

Note that paths that connect the light source directly to the camera are missing. In real time applications, (primary) light sources are typically paired with some geometry that represents the light source (for example, a model of a lamp). This geometry is shaded in a way that gives the appearance of emitting light, even though it technically does not. The light source itself is invisible.

We need to take into account that light along all paths (including paths of type A) travels through a medium. The medium absorbs and scatters light. The attenuation from absorption and scattering is described by Beer's Law[4]:

$$I(s) = I_0 \, e^{-\beta \, s}.$$

The light's initial intensity is described by I_0, and $I(s)$ is the intensity after traveling s length units in a medium with an extinction coefficient β (see below). Beer's Law has been around since the 18:th century, more recently, Pharr and Humphreys [2010] derived and described it in the context of computer graphics.

The amount of attenuation depends on the properties of the medium, specifically, the medium's extinction coefficient. The extinction coefficient describes the amount of light that is lost in the medium. Extinction occurs through absorption and scattering (the extinction coefficient is the sum of the absorption and scattering coefficients). In reality, media have different amounts of extinction at different locations (due to variations in e.g., the medium's density). Because of this, the extinction coefficient would typically be a function of position.

A common approximation in real-time rendering is the assumption that a medium is homogeneous. A homogeneous medium has the same properties everywhere, and, for instance, the extinction coefficient becomes a constant. Examples where this approximation can be valid include fog and smog (Figures 2a and 2b). Smoke and clouds are examples where this approximation typically is invalid, and that need to be treated as heterogeneous media instead (Figure 2c).

A second property of the medium is the scattering phase function. The phase function specifies how much light is scattered into a certain direction relative to the direction of the incoming light. In the simplest case, isotropic scattering, the phase function equals a constant ($\frac{1}{4\,\pi}$) and light is scattered equally in all directions. Nishita et al. [1987] discuss other, more advanced, phase functions.

Single Scattering. We first focus on single scattering, specifically paths of type B_1 (Figure 5b). The amount of light that reaches the camera along a single view ray is

$$L = L_d + L_a,$$

where L_d is the contribution from paths of type A, and L_a is the contribution from paths of type B_1 (Figure 6). Nishita et al. [1987] introduce the airlight integral that computes L_a:

$$Ai(\mathbf{r}; u, v) = I_0 \int_u^v \beta \, k(\alpha) \, \frac{e^{-\beta \, d(\mathbf{r};t)}}{d(\mathbf{r};t)^2} \, e^{-\beta \, t} \, dt.$$

[4]Beer's Law is also known as the Beer-Lambert Law, or as Bouguer's Law (the name used by Nishita et al. [1987]), or by a combination of these three names. Pharr and Humphreys [2010] omit the name altogether and simply derive it from first principles.

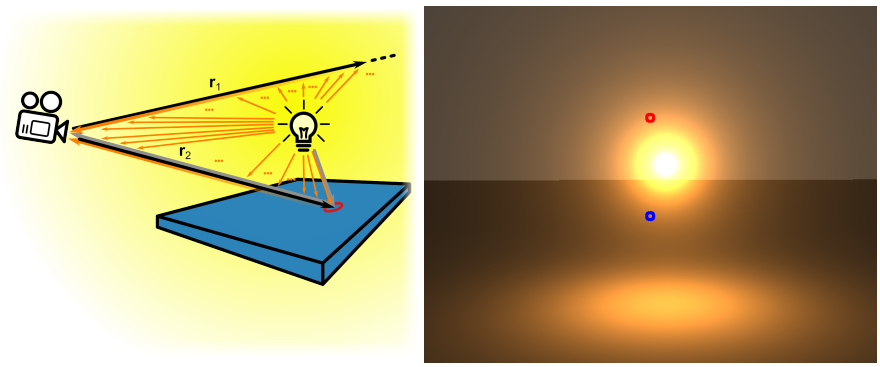


Figure 6. Airlight integration along two view rays. The left figure shows a schematic overview. Ray $\mathbf{r}_1$ (top) extends from the viewer to infinity, and therefore its airlight contribution (orange arrows) equals $Ai(\mathbf{r}_1; 0, \infty)$. The ray never intersects with a surface, so $L = L_a$. Ray $\mathbf{r}_2$ (bottom) extends from the viewer to its intersection with the surface. For this ray, $L = L_d + L_a$, where L_d is the (attenuated) direct illumination from the surface (gray arrows). The airlight contribution for $\mathbf{r}_2$ equals $L_a = Ai(\mathbf{r}_2; 0, x)$, where x is the distance from the viewer to the intersection with the surface. The right hand image displays a rendering of such a scene, as seen from the camera. Example locations that correspond to the situation in the left hand figure are highlighted with a red ($\mathbf{r}_1$) and blue ($\mathbf{r}_2$) circle, respectively.

Here, $\mathbf{r}$ describes the ray along which integration is performed, and the scalar parameters u and v define a limited interval on this ray. In Figure 6, one view ray extends into infinity, so we would use $L_a = Ai(\mathbf{r}; 0, \infty)$, and one view ray is terminated by the surface it is intersecting ($L_a = Ai(\mathbf{r}; 0, x)$, where x is the distance to the intersection). The function $d(\mathbf{r}; x)$ computes the distance from the point x on the ray $\mathbf{r}$ to the light source. The medium is described by the parameters β (the extinction coefficient) and $k(\alpha)$ (the scattering phase function).

Several papers study the airlight integral and propose efficient ways of its evaluation. Sun et al. [2005] present a GPU friendly method for evaluating $Ai(\cdot)$ that relies on a pre-computed texture. Their method, however, assumes isotropic scattering ($k(\alpha) = \frac{1}{4\pi}$). We employ a slightly modified variant of this method in both Paper **II** and Paper **III** and therefore limit ourselves to isotropic scattering (details on the modified variant can be found in Eisemann et al. [2011, Section 9.3.1]). An interesting note is that Sun et al. [2005] also consider other paths of type B_i and describe methods to take effects from these paths into account during rendering. These methods ignore scene geometry, however.

Paper **II** makes few assumptions about the method used to evaluate the airlight contributions. We could replace the evaluator based on the work of Sun et al. [2005] with one based on work by e.g., Pegoraro et al. [2009, 2011] (who demonstrate airlight evaluation with some forms of anisotropy).

Multiple Scattering. To compute multiple scattering, we rely on radiative transfer methods. The seminal work of Chandrasekhar [1960] explores these methods in some detail[5]; Pharr and Humphreys [2010] include excellent descriptions of this, as does the survey by Cerezo et al. [2005].

Beer's Law is derived from the differential equation $dI(s) = -\beta I(s)\, ds$. It describes how, at each point in space, a small fraction of the intensity $I(s)$ is absorbed or scattered away. The differential form of Beer's Law is a part of the transport equation that describes light propagation in a medium [Arvo, 1993]:

$$\omega \cdot \nabla L(\mathbf{x}, \omega) = -\beta L(\mathbf{x}, \omega) + \sigma \int p(\omega, \omega')\, L(\mathbf{x}, \omega')\, d\omega'.$$

Here, ω represents a direction and $\mathbf{x}$ a position in space. The coefficients β and σ describe extinction and scattering, respectively. Radiance $L(\mathbf{x}, \omega)$ replaces the one-dimensional intensity $I(x)$ from Beer's Law. The function $p(\omega, \omega')$ is a more general form of the scattering phase function $k(\alpha)$. Compared to Beer's Law, the transport equation additionally considers in-scattering. Note that emission is omitted in this description, as we do not consider media that emit light in Paper **III** (large scale homogeneous media that emit light are not very common in the everyday world).

The above model can be summarized as follows: the change in radiance in a certain direction ω at each point $\mathbf{x}$ in space is computed from the sum of the incident radiance, attenuated by absorption and out-scattering; and the in-scattered radiance into direction ω. With this description we can see that, conceptually, scattering changes the direction of a portion of the radiance into directions ω' to ω. Additionally, some radiance is lost due to absorption.

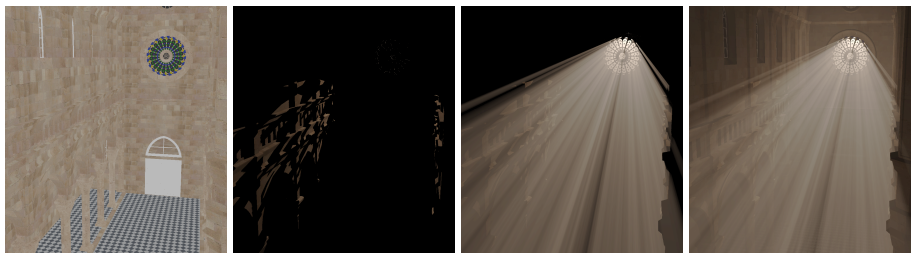
Paper **III** presents a propagation scheme based on the this model.

Visual Impact of Scattering. In a model that only supports paths of type A, surfaces that face away from a light source and surfaces that are occluded (i.e., in shadow) never receive any illumination (Figure 7b). Adding paths of type B_1 does not change this, but instead makes the participating medium visible (Figure 7c). However, adding paths of type B_2 and C_2 adds indirect illumination to surfaces that are otherwise completely shadowed (shown in Figure 7d and illustrated in Figures 8a and 8b). Note that Figure 7d also includes the remainng types of light paths B_i and C_i. In particular, paths of type C_1 make the medium visible in regions of space that are not directly illuminated (illustrated in Figure 8c).

3.1 Paper II

Problem. By evaluating the airlight integral, we can compute the amount of light that an illuminated medium scatters towards the viewer (paths of type B_1 in Figure 5). However, not all of the medium is illuminated, as objects in the scene

[5]While important, the work of Chandrasekhar [1960] is perhaps not the most approachable one on the topic of radiative transfer methods, at least not for computer graphics applications.



(a) (b) (c) (d)

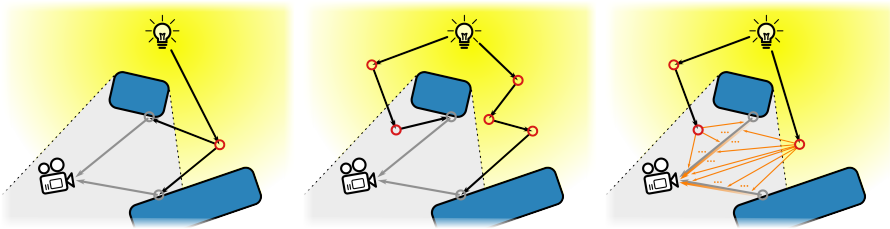
Figure 7. Renderings of the Sibenik Citadel Model with different sets of light paths. The leftmost image (a) provides a reference rendering with no lighting applied. A light source is located just outside the circular window in the top of the view. In (b), only direct illumination (paths of type A) are considered. A significant part of the scene is now in shadow. Adding volumetric shadows (paths of type B_1, using the technique from Paper **II**) makes the illuminated parts of the medium visible (c), and gives the viewer information about the location of the light source. No new surfaces are revealed, however. When considering multiple scattering (using the technique from Paper **III**), previously hidden surfaces are now illuminated thanks to the inclusion of light paths of type B_2 and C_2 (d). Remaining paths of types B_i and C_i are also present in the last image.

block the light and thereby cast shadows not only onto surfaces, but into the medium itself. We want to capture the latter effect, where shadows become visible in the participating medium, creating a volumetric shadowing effect.

Each view ray now potentially contains several distinct intervals that are illuminated and therefore contribute airlight. Figure 9 illustrates one view ray with three such illuminated intervals. Each interval is defined by a position t_l^i, where the view ray enters an illuminated region, and a position t_s^i, where the view ray leaves the illuminated region.

The core problem is to efficiently identify these intervals and then to evaluate each interval's airlight contribution.

One class of previous methods employ ray-marching or slicing, where a contribution is computed at discrete sample-positions along each view ray [Dobashi et al., 2002]. Ray-marching is very flexible (for example, Wyman and Ramsey [2008] and Engelhardt and Dachsbacher [2010] support textured light sources), but tends to require many samples (many techniques focus on reducing the number of samples that are required - this includes both previously mentioned techniques and also, for example, Toth and Umenhoffer [2009] and Imagire et al. [2007]). A different approach relies on shadow volumes. Shadow volumes avoid issues with sampling. However, traditionally, these methods suffer from drawbacks such as requiring expensive (and sometimes ambiguous) sorting operations [Venceslas et al., 2006], or requiring costly depth peeling [James, 2003]. Our method is also based on shadow volumes, but avoids these problems.



(a) (b) (c)

Figure 8. Light paths with special visual impacts. Single scattered light from paths of type B_2 and multiple scattered light from paths of type C_2 can illuminate surfaces that are in shadow or that are facing away from the light source (a and b). Additionally, paths of type C_1 add scattering in shadowed regions, and thereby make the medium visible in those locations (c).

Methodology. A fundamental property of integration is the ability to split a (finite) integral $\int_a^b$ into the sum $\int_a^c + \int_c^b$. We initially realized that this enables us to reformulate the problem of integrating an airlight-contribution for each interval $\left[t_l^i, t_s^i\right]$ on a ray $\mathbf{r}$ into a problem where we compute an airlight-contribution for each boundary:

$$\begin{aligned}\sum_i Ai\left(\mathbf{r}; t_l^i, t_s^i\right) &= \sum_i Ai\left(\mathbf{r}; t_v, t_s^i\right) - \sum_i Ai\left(\mathbf{r}; t_v, t_l^i\right) \\ &= \sum_j s_j \, Ai\left(\mathbf{r}; t_v, t_b^j\right)\end{aligned}$$

Here, $t_v = 0$ is the camera's position on the view ray $\mathbf{r}$ (although any fixed point on the view ray would work), and t_b^j is the set of all boundaries (i.e., the union of all t_l^i and t_s^i). The sign s_j depends on whether t_b^j is a transition from lit to unlit space or vice-versa. Furthermore, addition is commutative, and we can accept boundaries t_b^j in any order.

We found traditional shadow volumes to be problematic, because of overlapping volumes. Instead, we decided to derive the volumes from shadow maps, similar to the method described by McCool [2000]. The resulting polygonal volumes are free from overlaps and enclose the regions of the scene that are directly illuminated.

We implemented the method using OpenGL and GLSL shaders. We derived our airlight evaluator from the method described by Sun et al. [2005]. We evaluated performance for a varying number of shadow map resolutions. For high-resolution shadow maps, we explored a dynamic tessellation scheme which reduces the amount of geometry used to represent the shadow volumes.

Algorithm Overview. Using a shadow map, we construct a simply-connected polygonal volume that encloses the space directly illuminated by the light

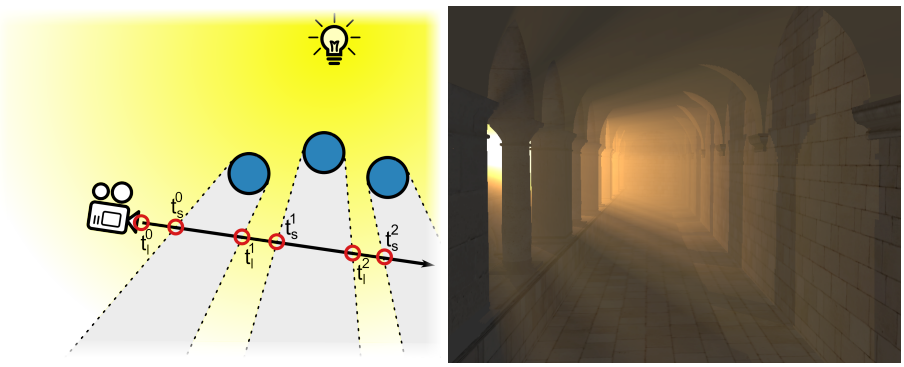


Figure 9. Left: The depicted view ray includes airlight contributions from three separate intervals. Each interval begins at a transition to lit space, t_l^i, and ends at a transition from lit to shadowed space, t_s^i. Right: Rendering of a view where several pillars give rise to multiple separate intervals with individual airlight contributions.

source. We render this polygonal volume with neither depth-testing nor back face culling, such that all surfaces intersecting any view ray generate fragments. We compute the airlight contribution for each fragment; each fragment represents one boundary t_b^j. The type of boundary (entering or leaving a lit region), and therefore the boundary's contribution's sign, is determined from the GLSL attribute `gl_FrontFacing`. All contributions are accumulated into the frame buffer.

Contributions. Paper **II** presents a simple, but yet efficient method that can capture volumetric shadows from single scattering in homogeneous and isotropic media at real-time frame rates.

The base algorithm only requires support for basic shaders and the ability to accumulate results into a floating-point frame buffer. It is therefore viable on a wide range of graphics hardware.

Unlike sampling-based methods, our method produces exact results with respect to the shadow map resolution.

3.2 Paper **III**

Problem. Paper **II** considers paths of type B_1 (Figure 5). We now want to develop a method that supports additional types of light paths.

At the very least, we want to support paths of type B_2, as this enables indirect illumination of surfaces that would otherwise receive no light. Next, we would like to support paths of type C_1, since this enables light to interact with the participating medium outside of the directly illuminated regions of space (where single scattering occurs). We considered these two types of paths most important,

but would of course like to support additional types of paths (and, the final method does in fact support all paths of type B_i and C_i).

Methodology. We separate volumetric shadows that originate from single scattering from other types of scattering. Volumetric shadows include high-frequency effects, and several dedicated techniques attack this problem successfully (for example, Paper **II** and also later work by Baran et al. [2010]; Chen et al. [2011] and Wyman [2011]).

Other types of scattering produce much lower frequency effects. We simulate these effects using *light propagation volumes* (LPV) described by Kaplanyan and Dachsbacher [2010]. For this, we developed a modified propagation scheme that takes scattering into account. Further, we explored the use of the information from single scattering to seed the LPVs in order to reduce the number of simulation iterations and increase the accuracy of the solution.

We implemented the propagation in LPVs using CUDA. For other tasks, we used OpenGL. We test our proof-of-concept implementation on an NVIDIA GTX 480 GPU. We compare our renderings to solutions produced by an off-line renderer.

Algorithm Overview. We start by seeding the LPV. We use light reflected from surfaces (as described by Kaplanyan and Dachsbacher [2010]) and light scattered once by single scattering. We find both contributions in a single *reflective shadow map* (RSM). We derive fuzzy blockers from the same RSM. If needed, we insert additional fuzzy blockers by rendering supplementary (plain) shadow maps.

Next, we propagate light using our modified propagation scheme. This is an iterative process, and for grids of resolution 32^3 we perform eight iterations.

We then render the geometry. We use the data stored in the LPV to compute indirect illumination for rendered surfaces in addition to the usual direct illumination. Next, we perform ray marching in the LPV to find approximate multiple scattering in the medium. Finally, we render the single scattering contribution using the method from Paper **II**.

Contributions. We present a real-time method for multiple scattering. Our method includes support for all light paths presented in Figure 5.

Separating single scattering from higher order effects enables the use of very coarse methods like LPVs, for which we present a formal propagation scheme that takes the participating medium into account. Our method retains features from the original LPV method by Kaplanyan and Dachsbacher [2010], such as the indirect surface-to-surface illumination (illustrated in Figures 10a and 10b).

4 Many-Light Shading

In the previous section, we described lighting resulting from one light source (but in the presence of a participating medium). In this section, we will now explore settings with many light sources (albeit without the participating medium). A

(a) (b) (c)

Figure 10. Indirect light paths. (a) Illustration of paths where light is emitted from a light source towards a first surface, from where it is reflected to a second surface and then towards the eye. LPVs were originally developed to enable indirect illumination from these types of paths. (b) Rendering with illumination from these paths and from direct light only. Note the indirect red illumination from the curtain visible on the arch. (c) Direct illumination only, for comparison.

trivial way to include multiple light sources is to simply repeat the computations for each light source and sum all contributions[6]. This becomes costly as the number of light sources increases. Our aim is to reduce this cost.

Given a single light source, and a function `light()` that computes the lighting from a light source for a view sample, we can express shading using the following pseudo code:

```
function shade( viewSample, lightData )
{
    outputColor = light( viewSample, lightData );
}
```

The naive approach to shading from many light sources results in the following code:

```
function shade( viewSample, numLights, lightData[] )
{
    outputColor = vec3( 0.0 );

    for( i = 0; i < numLights; ++i )
        outputColor += light( viewSample, lightData[i] );
}
```

The `light()` function is now called `numLights` times for each shaded view sample.

For a small number of lights, this simple approach works well. But for large numbers of lights, the amount of computations quickly becomes overwhelming: shading N view samples with M lights requires $M \times N$ evaluations of the `light()` method ($M \times N$ *lighting computations*). Many-light shading methods attempt to increase performance mainly by reducing this number.

[6] Although, in Paper **III**, we would rather inject contributions from all lights into the LPV and perform the simulation only once. Other steps, such as the seeding and final surface shading, need to be repeated for each light source, though.

In forward shading, the problem is potentially even worse. The computations of `shade()` are performed for every generated fragment. Overdraw can cause several fragments to be generated for each pixel, of which only one fragment affects the final image. Therefore, we would like to eliminate unnecessary computations due to overdraw.

Another way to reduce the number of lighting computations is by identifying which lights each view sample is affected by (or conversely, which view samples each light affects), and only perform lighting computations for this reduced set of view-sample–light pairs. Expressed in pseudo-code, we get the following:

```
function shade( viewSample, lightData[] )
{
    outputColor = vec3( 0.0 );

    numLightsAtSample = num_lights_affecting( viewSample );
    for( i = 0; i < numLightsAtSample; ++i )
    {
        lightIdx = global_light_index( viewSample, i );
        outputColor += light( viewSample, lightData[lightIdx] );
    }
}
```

Note that in the worst case (when all lights affect all view samples), this approach yields no improvement. However, in real-time rendering, light sources are commonly modelled with a limited range. This ensures that in a typical scene the worst case never occurs. For example, Ferrier and Coffin [2011] mention scenes in the game "Need For Speed: The Run" with approximately 2600 light sources (2200 spot lights and 400 point lights), but where less than 400 light sources are visible each frame. Further, they aim for an average of 6 light sources per view sample[7].

Deferred shading provides a solution for the problem of overshading due to overdraw. In deferred shading, geometry is rendered to a G-buffer in a first pass [Thibieroz, 2003][8]. The G-buffer stores all attributes required to compute shading, which is performed in a separate pass. In this second pass, we know which samples are visible (namely, the ones that were stored in the G-buffer) and therefore avoid unnecessary computations.

Thibieroz [2003] suggest drawing a full screen quad per light in the second pass. Instead, it is possible to draw smaller bounding volumes for each light [Hargreaves, 2004; Hargreaves and Harris, 2004], eliminating computations for view samples that are known to be out-of-range for the light. One drawback of either method is that data is repeatedly loaded from the G-buffer and results are repeatedly accumulated into the output color buffer. So, while reducing the amount of lighting computations, a massive use of memory bandwidth is introduced.

[7] Per tile, in fact, since they employ tiled shading.

[8] Technically, deferred shading was presented earlier (e.g., by Deering et al. [1988] and Tebbs et al. [1989]), but Thibieroz [2003] describes a modern implementation.

Expressed in pseudo-code, conceptually the following situation emerges:

```
function shade( GBuffer, viewSampleIdx, lightData[] )
{
    outputColorBuffer[viewSampleIdx] = vec3(0.0);

    for each light potentially affecting the current view sample
    {
        /* Note: each iteration here corresponds to a separate
         * fragment shader invocation. */
        viewSample = GBuffer[viewSampleIdx];

        lightIdx = current light;
        outputColor = light( viewSample, lightData[lightIdx] );

        outputColorBuffer[viewSampleIdx] += outputColor;
    }
}
```

Note that the loop's body in this version is spread across many fragment-shader invocations, since it is executed once per fragment generated when rendering light bounding volumes. From a memory-bandwidth point-of-view, we would much rather have the following situation:

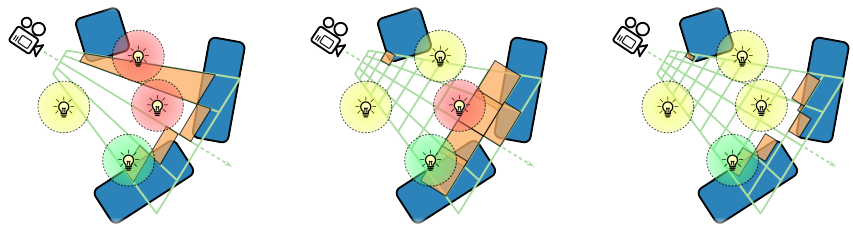
```
function shade( GBuffer, viewSampleIdx, lightData[] )
{
    viewSample = GBuffer[viewSampleIdx];

    outputColor = vec3(0.0);
    for each light potentially affecting the current view sample
    {
        lightIdx = current light;
        outputColor += light( viewSample, lightData[lightIdx] );
    }

    outputColorBuffer[viewSampleIdx] = outputColor;
}
```

In this version, we read the G-buffer only once, and store a result to the color buffer exactly once, thereby eliminating a large amount of memory traffic. On the other hand, we now need to know what lights affect a certain sample. Storing light-lists per sample is possible, as shown by Trebilco [2009], but expensive.

Tiled shading instead finds what lights affect samples in a tile (samples arranged in a 2D rectangle). Initial presentations of Tiled shading include Balestra and Engstad [2008] and Swoboda [2009]; the latter used tiled shading specifically to enable offloading the lighting computations from the GPU to *synergistic processing units* (SPU) available on the Playstation3's Cell processor. The technique quickly gained some traction in the game developer community [Andersson, 2009; Coffin, 2011; Ferrier and Coffin, 2011; White and Barré-Brisebois, 2011], and later in academia [Billeter et al., 2013; Harada et al., 2012; Olsson and Assarsson, 2011].



(a) (b) (c)

Figure 11. Comparison of tiled shading and clustered shading. Solid objects are shown in blue, and bounding volumes for tiles (a) and clusters (b and c) in orange. Green light sources affect surfaces (and are not culled), yellow lights are correctly culled, and red lights represent false positives (light sources that are not culled, but do not affect any geometry). The leftmost image (a) illustrates the fundamental problem with tiled shading. The topmost tile's bounding volume must stretch across a significant portion of the view's depth range to enclose all visible geometry intersecting the tile. The red light sources intersect with the tile's bounding volume and must be considered during shading, despite not affecting any visible geometry. Clustered with implicit bounds (b) already avoids the view-dependent stretched out bounding volumes. Computing explicit bounds for clusters further reduces false positives (c).

Paper **IV** investigates one of the fundamental problems with tiled shading, namely that grouping samples into 2D tiles is problematic in 3D scenes (Figure 11a).

4.1 Paper **IV**

Problem. In tiled shading, samples are grouped into 2D tiles. Depth bounds are then derived for each tile, which allows reconstruction of an *axis aligned bounding box* (AABB) for each tile. Given a regular frame buffer, this is very simple and efficient.

However, if the view samples within a tile cover a large depth range, the tile's bounding box must extend to enclose all view samples. The resulting AABB spans a large empty region, which leads to many false positives when assigning lights (Figure 11a), lowering overall performance and efficiency. The amount of lights assigned to each tile also becomes very view dependent [Ferrier and Coffin, 2011], which leads to unpredictable and uneven performance.

We aim to explore 3D and higher-dimensional groupings (clusterings) of view samples to compute a better light to view sample mapping. By doing so, we aim to increase the number of light sources that can be used in any scene and also increase robustness by making performance less view-dependent and more predictable.

Methodology. We extend the concept of tiles into clusters, which group view samples in three or more dimensions. Specifically, we consider grouping view samples on their 3D position (resulting in a 3D grouping), and on their 3D position and normal (resulting in a 5D grouping). Each cluster is uniquely identified by a corresponding cluster key.

The number of clusters is much larger than the number of tiles. Combined with our aim to support even more light sources, we need to improve light assignment. We therefore explore a hierarchical approach to light assignment.

We implement the above using OpenGL and CUDA. We make extensive use of GPGPU techniques from Paper **I**. We compare efficiency of our method and its variations to other algorithms (traditional deferred shading and tiled shading) by measuring the number of lighting computations under varying conditions. We measure performance for various steps in the clustered shading pipeline.

Algorithm Overview. A cluster key is computed for each view sample based on the view sample's 3D position and optionally on its normal. View samples with identical cluster keys reside in the same cluster; this creates a 3D (or optionally 5D) grouping of view samples. Extracting unique cluster keys from all view samples yields a list of active clusters.

Lights are assigned to each cluster by constructing and traversing a bounding volume hierarchy constructed from the lights' bounding volumes. View samples are shaded by (re-)computing the cluster key, accessing the lights assigned to that cluster (through the cluster key), computing lighting from each assigned light and accumulating the results.

Contributions. We present a simple exponential clustering based on the 3D position of a view sample. We also present an extended 5D clustering based on the 3D position and normal of a view sample. We explore several variations of each clustering method where we use implicit (Figure 11b) and explicit (Figure 11c) bounding volumes for both positions and normal cones. We present two methods to extract active clusters from the depth buffer, and optionally from normals stored in a G-buffer. The first method is based on sorting view samples, and the second method uses scheme inspired by page-tables from virtual memory management.

For light assignment, we present a CUDA-based algorithm where we construct a bounding volume hierarchy over the light sources and then traverse the clusters against this bounding volume hierarchy. Both traversal and construction is performed each frame. We show that our light assignment method scales well up to at least 1M light sources (compared to previous methods that handle a few thousands to perhaps ten thousand light sources).

We find that (for our test scenes) the 3D clustering performs best. The more complicated clustering methods reduce the number of lighting computations further. In fact, the 5D clustering enables light culling based on the facing of

surfaces, which can reduce the number of lighting computations beyond what methods that compute pixel-exact light volume overlap can achieve. Unfortunately, in our implementation and our test scenes, the overheads associated with constructing the more complex clustering offset the gains in shading time.

We show that clustered shading produces a much more predictable performance when compared to tiled shading. The tiles' bounding volumes are very view dependent, which results in large variations in the number of lighting computations (and, by that, in shading time). Clustered shading is much less view dependent, as the variations in the number of lighting computations are much smaller.

We demonstrate clustered shading with both forward and deferred shading.

4.2 Paper V

Problem. In this paper, we aim to bring modern and high-end graphics to mobile devices. Previously, two main approaches were considered for mobile graphics: either off-loading rendering completely to a remote server ("the cloud"), and streaming the resulting video to the mobile device; or rendering everything locally. The former approach ignores the rapidly increasing capabilities of mobile GPUs (and mobile devices in general). The latter approach makes no use of the always-connected nature of mobile devices.

The primary goal was to identify what (rendering) work could easily be off-loaded to a remote machine given the following conditions:

- The system should be resistant to network interruptions.
- The server should scale well with an increasing number of clients.
- The system should be able to adapt to different on-client capabilities.
- Network traffic should remain as minimal as possible.

Our target system is a mobile tablet. We use, for example, the Nexus 10 device, which supports OpenGL|ES 2.0 (and more recently OpenGL|ES 3.0) via the ARM Mali T-604 GPU. However, we would like to potentially scale up quality for more capable clients.

A target application is rendering for 3D navigation software, where rendering must not be interrupted because of network conditions[9]. This also eliminates plain video streaming from our options.

There are several reasons for keeping network traffic minimal. For one, bandwidth might be limited due to network coverage. Next, users might have limited alloted data budgets. Finally, other data might need to be transferred too (e.g., downloading new maps for areas not yet available on the device).

[9]The idea being that you probably really want your navigation software to work when you're so far from civilization that you're losing cell coverage. Also, at that point you might care less about fancy graphics, so being unable to show anything due to this is not an acceptable excuse.

Methodology. Our initial investigation focused on finding a good conceptual split between local and remote work. We found that indirect illumination is a good candidate for off-loading to a remote server:

- Indirect illumination can be disabled with and replaced with reasonable fall-backs (e.g., increased ambient light) in case of network problems.
- A single indirect illumination solution is valid for all clients that view the same scene at the same 'time'.

Several representations of indirect illumination were considered. We settled for an approach based on *virtual point lights* (VPL), where indirect illumination is represented using spot lights. Crassin et al. [2013] also mention off-loading of indirection illumination comptutations to remote server, but explore different representations. For mobile devices, they mainly propose a video-streaming solution.

On the client side, we implemented first a tiled forward shading renderer [Billeter et al., 2013] targeting OpenGL|ES 2.0. Later, we adapted clustered forward shading [Olsson et al., 2012] to work with our target devices and implemented a renderer targeting OpenGL|ES 2.0 and OpenGL|ES 3.0 that utilizes the modified clustered shading scheme. The adapted clustered forward shading scheme computes light assignment up-front on the CPU, which was shown to be possible by Persson and Olsson [2013]. This avoids the dependency on GPGPU facilities. We implemented the renderer on both PC (for more rapid development) and on the target mobile device.

We implemented a simple server that sends light sources to the client. For selected light sources, the server computes additional spot lights that approximate indirect illumination. The server keeps track of changes to the light sources and only generates updates to clients when the light sources change. Therefore, static lighting uses extremely little bandwidth.

We measured performance on both the server and the client. On the server, we compared static and per-client work. On the client, we measured rendering performance under varying lighting conditions.

Algorithm Overview. The implementation is split into two parts: the server and the client. The client tries to maintain a connection to the server at all times.

The server maintains list of light sources and tracks all connected clients. Each server-side frame, the server updates all light sources, keeping track of what properties have changed during the update. Pending updates are accumulated for each client. Minimal sets of changes are transmitted to clients when they are ready to receive an update, as determined from network conditions.

On the client, received updates are integrated into the client's light list. Clients render the scene locally, using a modified clustered forward shading scheme, where light assignment is done upfront. This requires assigning lights into all

potential clusters; in order to minimize the work load, a cascaded clustering that reduced the total number of clusters is used.

Contributions. We present a rendering system that is capable of utilizing both client hardware and remote servers. Our system allows sharing of server-side computations among many clients, which enables the server to scale well with respect to the number of connected clients.

On the client side of this system, we present and evaluate a modified clustered shading scheme that works with current mobile hardware and thereby enables multi-light rendering on our target devices. We implement and evaluate the scheme on the Nexus 10 tablet.

5 Discussion and Future Work

So far, my thesis has focused on presenting background to the techniques presented in the included papers, and on summarizing the papers themselves. In this section, I will describe and discuss findings and ideas that are related to the presented methods, but that are not included in the papers. For example, Section 5.1 presents two further developments to the core technique of clustered shading (Paper **IV**). We discovered these advances after the initial publication and presentation of our clustered shading technique – both developments are also presented in later publications.

5.1 Recent Advances to Clustered Shading

The interest for tiled and clustered shading has been enormous. On one hand, several persons from the games industry have expressed interest in clustered shading for their future products[10]. On the other hand, our research group has continued research related to clustered shading, which also has resulted in several spin-off ideas. For example, Sintorn et al. [2014] cluster view samples and then build a hierarchy of clusters in order to accelerate per-triangle shadow volumes. Olsson et al. [2014] use clustered shading together with virtual shadow maps to enable many-light rendering with shadows.

More directly, we presented an extension to clustered shading that enables transparency and hardware anti-aliasing (MSAA) [Olsson et al., 2012]. This extension is also applicable to tiled shading [Billeter et al., 2013]. Further, Persson discusses an adaption of clustered shading for potential use in an upcoming game [Persson and Olsson, 2013].

Because of this continued work, we developed several modifications and potential improvements to the original clustered shading method. The important ideas are summarized as follows.

[10]Tiled shading was developed and employed in the instustry (e.g., Balestra and Engstad [2008]) before appearing in academic publications Harada et al. [2012]; Olsson and Assarsson [2011].

Supporting transparency. Paper **IV** mentions forward shading, but the presented method does not support transparency (out-of-the-box). In Paper **IV**, the clusters are always extracted from the depth buffer, regardless of whether forward or deferred shading is employed. (The initial implementation of clustered forward shading required a pre-Z pass.)

With transparent objects, it is no longer possible to extract clusters from the depth buffer. Each pixel might contain contributions from several fragments, each with an individual depth. The depth buffer can only store a single value per pixel, which makes it impossible to recover depths for all contributing fragments.

We solve this by using image writes (`imageStore()` in GLSL) in the fragment shader. The method consists of the following steps:

1. Initialize buffer to zero.
2. Render geometry. In fragment shader, compute cluster key and use `imageStore()` to write a non-zero number to the buffer at that index.
3. Extract non-zero elements from buffer. The locations of the non-zero elements identify which clusters contain fragments.

The buffer can either be a dense buffer, where each cluster is statically allocated one element, or the page-table based method presented in Paper **IV** can be used if the maximal number of clusters is large.

Typically, using e.g. `imageStore()` would disable early fragment tests (because the shaders now have a side effect), which may have performance implications. However, we are not interested in fragments that are completely occluded (as determined with depth testing), so it is safe to enable early fragment tests in the shader (and doing so is actually beneficial for both clustering and shading performance). Because of this, opaque geometry should be rendered first in a rough front-to-back order, followed by transparent geometry (which must typically be rendered back-to-front).

Several instances of the fragment shader may attempt to use `imageStore()` to the same pixel in the image concurrently. Normally, this would indicate the presence of a race condition. In practice, one of the `imageStore()` operations will win, which results in a non-zero value at that pixel – this is sufficient for us.

With this method, it also becomes trivial to support hardware-accelerated anti-aliasing schemes like MSAA.

Explicit Bounding Volumes. The page-table based clustering presented in Paper **IV** and the method using `imageStore()` described above both have one common drawback, compared to the sorting-based methods from Paper **IV**. We cannot compute the explicit bounding volume of a cluster (we do not keep track of which fragments contributed to each cluster) but must use the less precise implicit bounds. The explicit bounds are often much smaller, which leads to fewer false positives during light assignment.

In Section 4.4 in Olsson et al. [2014], we present a method to compute tighter bounds for clusters that is compatible with both the page-table based approach and with the `imageStore()`-based method. We subdivide each cluster into a 10^3 grid. For each view sample, we compute the coordinate in this 10^3 grid of the cluster in which the view sample resides. We then set the bit corresponding to each coordinate in three 10-bit fields. The three bit-fields can be packed into a single 32-bit integer, which is combined with other such integers using bitwise `or` operations (we need to ensure that the bitwise-`or` is atomic). In the `imageStore()`-based method, the call to `imageStore()` is, for example, replaced with a call to `imageAtomicOr()`[11].

Using bitwise operations (count leading/trailing zeros), we can extract a tighter bounding volume from the resulting integer. This method requires 32-bits of storage per (active) cluster. Additionally, the hardware must support a relatively high throughput of atomic operations.

5.2 Tiled Shading on Mobile Devices

Section 4.2 mentions that we first attempted to employ tiled shading on the client-side renderer. We later switched to a renderer based on clustered shading. Paper **V** does not comment on this further. This section summarizes our findings from implementing tiled shading in OpenGL|ES 2.0 and our reasons for ultimately switching to clustered shading.

We initially decided to use tiled shading because of its simplicity. Mobile devices lacked the GPGPU-capabilities required by the original clustering algorithm[12]. At that point, we did not consider performing light assignment on the CPU to a dense cluster structure to be a viable method (it remains one of the very performance-sensitive parts of the client-side renderer in Paper **V**).

The general advantages of clustered shading have already been discussed in this thesis. Because of the high relative cost of computing shading for one light source on the mobile devices, the problem with stretched-out tiles in tiled shading (to which many lights are erroneously assigned) becomes very noticeable. We did observe stretched-out tiles frequently in our test scenes.

Our tiled shading implementation performed the following steps:

1. Render geometry to frame buffer with a single depth buffer attachment.
2. Compute per-tile minimum and maximum view sample depths using GLSL and pack results into a 8-bit RGBA texture.
3. Copy per-tile min/max depths to host memory.

[11] In CUDA, we use `atomicOr()`. Since we're not interested in the return value, the compiler can emit a "reduction operation" (`RED.E.OR`) instead of the less efficient `ATOM.E.OR`.

[12] Both OpenGL image resources and OpenGL-OpenCL interop were unavailable. OpenCL further lacks meta-programming capabilities, which makes implementing e.g. efficient and general sorting methods difficult and extremely time-consuming.

4. Perform light assignment on CPU. Each tile needs to store a starting offset into a buffer containing the light data, and the number of lights that were assigned to this tile.
5. Pack per-tile information into an 8-bit RGBA texture. Pack light data into several textures (light position and radius were packed into two 8-bit RGBA textures, light color was stored in one 8-bit RGB texture, and spot light direction and opening angle used one 8-bit RGBA texture).
6. Render geometry to frame buffer with color buffer and depth buffer attachments. Optionally re-use depth buffer from Step 1.

Step 3 copies data from OpenGL to system memory. In OpenGL|ES 2.0, no asynchronous methods are available, and we ended up using the `glReadPixels()`-method. This introduces a stall. The copy might have been avoidable through extensions such as `OES_EGL_image_external`, but avoiding the stall is trickier.

We additionally required two geometry passes (Steps 1 and 6). These did (suprisingly) not represent a large bottleneck, even for moderately-sized models such as the Crytek Sponza. In the Desktop version of the client, we would reuse the depth buffer from the first pass to improve efficiency of early fragment tests. Due to the deferred architecture commonly used on mobile GPUs, this did not improve performance on the mobile devices, and we did not use the feature there.

In our implementation, Step 2 proved to be an unexpected bottle-neck. One of the difficulties was that the depth buffer is only accessible through texture-fetches in shaders; it cannot be copied to host memory directly. In OpenGL|ES 2.0, this required repacking the data into 8-bit textures. Further optimization might have lessened this problem, however. Also, personal communication with an ARM-engineer indicated that the hardware already tracks min- and max-depths for the tiles used in the hardware's deferred architecture. It might be possible to gain access to this data through an OpenGL-extension in the future.

Step 2 introduced one additional problem. OpenGL|ES 2.0 does not support access of individual samples from multi-sampled depth textures (in fact, it does not support accessing individual multi-samples at all; even multi-sampled color render targets are resolved before the corresponding texture is accessed in the shader). As a consequence, we cannot compute the correct min-/max-depth bounds if MSAA is used. A workaround is to avoid MSAA in Step 1 and only enable it in Step 6. This results in some visual artefacts, however.

OpenGL|ES 3.0 improves the overall situation somewhat. For instance, packing various values into 8-bit RGBA textures can be avoided. Support for uniform blocks and buffers is also an overall improvement. Many of the more fundamental problems remain, though – and our modified clustered shading method bypasses these issues.

Tiled shading should still be considered a valid technique under some circum-

stances. For instance, most of the above problems are related to computing min-/max-depth bounds for each tile. This could be avoided, for example, in application where top-down views with little variation in depth dominate. In that case, the light assignment can be done up-front on the CPU (similar to the up-front light assignment to clusters in Paper **V**), with the additional advantage that there are generally much fewer tiles than clusters (Section 4.8 in Billeter et al. [2013] further discusses the pros and cons of tiled shading).

5.3 Future Directions & Conclusion

Our clustered shading method has gathered some attention in the industry, and may, in fact, appear in at least one future high-profile game [Persson and Olsson, 2013]. The method seemingly attacks an important problem. Besides the improvements and follow-up methods that already are published, there are several additional potential directions that are open for exploration. Personally, I would like to revisit the clustering that additionally considers normals.

In our original publication, the overhead from the normal clustering offsets the gains in time spent shading. On one hand, the situation may look different with some of our recent improvements (i.e., those discussed in Section 5.1). On the other hand, normal clustering might perform better in certain situations. For one, a scene with many large light sources will benefit little from clustering on positions only - the lights' bounding volumes overlap with significant parts of the scene. Normal clustering would still be able to reduce the lighting computations. Furthermore, we currently approximate the lights' influence regions with spheres (an approximation that is common in real-time rendering). This breaks down in the presence of highly specular materials, where the highlights can appear cut off towards the outer limit of the bounding volume. Normal clustering might be able to take specularity into account and enable identification of clusters that are affected by specular contributions from a light source.

Other types of clusterings might also be interesting. It is conceivable to construct clusters from, for example, material properties rather than positions and normals. However, we have not yet identified a strong use case for doing so.

Rendering in the presence of a participating medium remains an interesting and open topic. Several recent publications (e.g., by Klehm et al. [2014]; Wyman and Dai [2013] present improved methods for rendering single scattering. Real-time multiple scattering and light transport seem to have received less attention.

I would like to explore the use of a sparse hierarchical representation for the LPV. The goal is to reduce the memory requirements and the number of computations during propagation by limiting the total number of cells in the volume. It might be possible to identify cells in a manner similar to the positional clustering method. (Olovsson and Doggett [2013] explore the use of a full octree to replace cascades [Kaplanyan and Dachsbacher, 2010] in the original LPV-setting without a participating medium. This might be a starting point for future investigation.)

In my introduction, I mention how GPUs have become increasingly flexible and powerful. One measure that my colleagues and I tend to reevaluate whenever a new GPU appears, is the measure of floating point operations per byte of available memory bandwidth. For instance, the NVIDIA GTX 280 GPU that I worked with at the beginning of my Ph.D. studies (the GPU was released in June 2008) has theoretical peak throughputs of approximately 930 GLFOPS and 140 GBytes/sec. This gives around 6.2 floating point operations per byte of memory bandwidth. For NVIDIA GTX Titan shown in Figure 1a this figure is instead 14.6. Although the exact number tends to fluctuate quite a bit, we have observed a steady rise in it.

I have followed mobile GPUs significantly less diligently, but a similar trend seems to hold true. For example, just comparing the various Exynos SoCs displays this trend already, where memory bandwidth has roughly tripled in the time period from 2011 to 2013. In the same period, theoretical computational throughput has increased more than $10\times$, from around 11 GFLOPS to about 140 GFLOPS.

As a consequence, algorithms that conserve memory bandwidth and rely more on arithmetic computations and on the increasing flexibility available on modern GPUs, should scale well in the future. Further, it seems that algorithms that work well in GPU-like environments are becoming increasing important. CPUs already are contain multiple cores, and are gaining wider SIMD. Additionally, all high-end smartphones and tablets sport relatively powerful GPUs. If we want to fully utilize the power of these (and future) devices, we will have to rely on increasingly parallel algorithms that are capable of running in GPU-like environments.

References

ANDERSSON, J. 2009. Parallel graphics in Frostbite – Current & Future. SIGGRAPH: Beyond Programmable Shading. URL http://dice.se/publications/parallel-graphics-in-frostbite-current-future/. 21

ARVO, J. 1993. Transfer equations in global illumination. In *Global Illumination, SIGGRAPH '93 Course Notes*, volume 42. URL http://www.cs.unm.edu/~jmk/arvo-notes-93.pdf. 14

BALESTRA, C. and ENGSTAD, P.-K. 2008. The technology of Uncharted: Drake's Fortune. Game Developers Conference. URL http://www.naughtydog.com/docs/Naughty-Dog-GDC08-UNCHARTED-Tech.pdf. 21, 26

BARAN, I., CHEN, J., RAGAN-KELLEY, J., DURAND, F., and LEHTINEN, J. 2010. A hierarchical volumetric shadow algorithm for single scattering. In *ACM SIGGRAPH Asia 2010 papers*, 178:1–178:10. ISBN 978-1-4503-0439-9. URL http://doi.acm.org/10.1145/1866158.1866200. 18

BILLETER, M., OLSSON, O., and ASSARSSON, U. 2013. Tiled forward shading. In *GPU Pro 4: Advanced Rendering Techniques*. A K Peters/CRC Press. ISBN 9781466567436. URL http://books.google.com/books?id=TUuhiPLNmbAC. 21, 25, 26, 30

BLELLOCH, G. E. 1990. Prefix sums and their applications. Technical Report CMU-CS-90-190, CMU School of Computer Science. URL https://www.cs.cmu.edu/~guyb/pubs.html. 8, 9

CEREZO, E., PEREZ-CAZORLA, F., PUEYO, X., SERON, F., and SILLION, F. 2005. A survey on participating media rendering techniques. *The Visual Computer*, 21(5):303–328. URL http://maverick.inria.fr/Publications/2005/CPPSS05/. 14

CHANDRASEKHAR, S. 1960. *Radiative transfer*. Dover Publications. ISBN 9780486605906. URL https://archive.org/details/RadiativeTransfer; http://books.google.com/books?id=CK3HDRwCT5YC. 14

CHEN, J., BARAN, I., DURAND, F., and JAROSZ, W. 2011. Real-time volumetric shadows using 1D min-max mipmaps. In *Symposium on Interactive 3D Graphics and Games*, 39–46. ISBN 978-1-4503-0565-5. URL http://doi.acm.org/10.1145/1944745.1944752. 18

COFFIN, C. 2011. SPU-based deferred shading in Battlefield 3 for Playstation 3. Game Developers Conference. URL http://dice.se/publications/spu-based-deferred-shading-in-battlefield-3-for-playstation-3/. 21

CRASSIN, C., LUEBKE, D., MARA, M., MCGUIRE, M., OSTER, B., SHIRLEY, P., SLOAN, P.-P., and WYMAN, C. 2013. CloudLight: A system for amortizing indirect lighting in real-time rendering. Technical report, NVIDIA Corporation. URL https://research.nvidia.com/publication/cloudlight-system-amortizing-indirect-lighting-real-time-rendering. 25

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., and HUNT, N. 1988. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, 21–30. ISBN 0-89791-275-6. URL http://doi.acm.org/10.1145/54852.378468. 20

DOBASHI, Y., YAMAMOTO, T., and NISHITA, T. 2002. Interactive rendering of atmospheric scattering effects using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 99–107. ISBN 1-58113-580-7. URL http://dl.acm.org/citation.cfm?id=569060; http://nis-ei.eng.hokudai.ac.jp/~doba/pub_doba.html. 15

DOU, H., YAN, Y., KERZNER, E., DAI, Z., and WYMAN, C. 2014. Adaptive depth bias for shadow maps. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 97–102. ISBN 978-1-4503-2717-6. URL http://doi.acm.org/10.1145/2556700.2556706; http://homepage.cs.uiowa.edu/~cwyman/pubs.html#AdaptiveSMBias. 11

EISEMANN, E., SCHWARZ, M., ASSARSSON, U., and WIMMER, M. 2011. *Real-Time Shadows*. Taylor & Francis. ISBN 9781568814384. URL http://www.realtimeshadows.com/. 13

ENGELHARDT, T. and DACHSBACHER, C. 2010. Epipolar sampling for shadows and crepuscular rays in participating media with single scattering. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 119–125. ISBN 978-1-60558-939-8. URL http://doi.acm.org/10.1145/1730804.1730823. 15

FERRIER, A. and COFFIN, C. 2011. Deferred shading techniques using Frostbite in "Battlefield 3" and "Need For Speed: The Run". In *ACM SIGGRAPH 2011 Talks*. URL http://doi.acm.org/10.1145/2037826.2037869. 5, 20, 21, 22

HARADA, T., MCKEE, J., and YANG, J. C. 2012. Forward+: Bringing deferred lighting to the next level. In *Proceedings of Eurographics 2012 - Short Paper*. URL http://dx.doi.org/10.2312/conf/EG2012/short/005-008. 21, 26

HARGREAVES, S. 2004. Deferred shading. Game Developers Conference. URL http://www.shawnhargreaves.com/DeferredShading.pdf. 20

HARGREAVES, S. and HARRIS, M. 2004. Deferred shading. 6800 Leagues Under The Sea. URL https://developer.nvidia.com/presentations-6800-leagues-under-sea. 20

HARRIS, M., SENGUPTA, S., and OWENS, J. D. 2008. Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*. Addison Wesley Professional. ISBN 9780321515261. URL http://books.google.com/books?id=ylNyQgAACAAJ; http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html. 9

HILLIS, W. D. and STEELE, JR., G. L. 1986. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183. ISSN 0001-0782. URL http://doi.acm.org/10.1145/7902.7903. 8, 9

HORN, D. 2005. Stream reduction operations for GPGPU applications. In *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*. Pearson Addison Wesley Prof. ISBN 9780321335593. URL http://books.google.com/books?id=QuBkQgAACAAJ; http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter36.html. 9

IMAGIRE, T., JOHAN, H., TAMURA, N., and NISHITA, T. 2007. Anti-aliased and real-time rendering of scenes with light scattering effects. *The Visual Computer*, 23(9-11):935–944. ISSN 0178-2789. URL http://dl.acm.org/citation.cfm?id=1283966; http://link.springer.com/article/10.1007/s00371-007-0140-9. 15

JAMES, R. 2003. True volumetric shadows. In *Graphics programming methods*, Charles River Media, Inc., 353–366. ISBN 1-58450-299-1. URL http://dl.acm.org/citation.cfm?id=957190; http://www.wavestate.com/pics/pocketmoon.pdf. 15

KAPLANYAN, A. and DACHSBACHER, C. 2010. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 99–107. ISBN 978-1-60558-939-8. URL http://doi.acm.org/10.1145/1730804.1730821. 18, 30

KLEHM, O., SEIDEL, H.-P., and EISEMANN, E. 2014. Prefiltered single-scattering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. URL http://graphics.tudelft.nl/Publications-new/2014/KSE14. 30

MCCOOL, M. D. 2000. Shadow volume reconstruction from depth maps. *ACM Trans. Graph.*, 19(1):1–26. ISSN 0730-0301. URL http://doi.acm.org/10.1145/343002.343006. 16

NISHITA, T., MIYAWAKI, Y., and NAKAMAE, E. 1987. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, 303–310. ISBN 0-89791-227-6. URL http://doi.acm.org/10.1145/37401.37437. 12

OLOVSSON, J. D. and DOGGETT, M. 2013. Octree light propagation volumes. *SIGRAD*. URL http://fileadmin.cs.lth.se/cs/Personal/Michael_Doggett/pubs/olovsson13-olpv.pdf. 30

OLSSON, O. and ASSARSSON, U. 2011. Tiled shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251. URL http://dx.doi.org/10.1080/2151237X.2011.621761. 21, 26

OLSSON, O., BILLETER, M., and ASSARSSON, U. 2012. Tiled and clustered forward shading. In *ACM SIGGRAPH 2012 Talks*. URL http://dx.doi.org/10.1145/2343045.2343095. 25, 26

OLSSON, O., SINTORN, E., KÄMPE, V., BILLETER, M., and ASSARSSON, U. 2014. Efficient virtual shadow maps for many lights. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 87–96. ISBN 978-1-4503-2717-6. URL http://doi.acm.org/10.1145/2556700.2556701. 11, 26, 28

PEGORARO, V., SCHOTT, M., and PARKER, S. G. 2009. An analytical approach to single scattering for anisotropic media and light distributions. In *Proceedings of Graphics Interface 2009*, 71–77. ISBN 978-1-56881-470-4. URL http://dl.acm.org/citation.cfm?id=1555880.1555902. 13

PEGORARO, V., SCHOTT, M., and SLUSALLEK, P. 2011. A mathematical framework for efficient closed-form single scattering. In *Graphics Interface*, 151–158. ISBN 978-1-4503-0693-5. URL http://www.cs.utah.edu/~vpegorar/research/2011_GI/. 13

PERSSON, E. and OLSSON, O. 2013. Practical clustered deferred and forward shading. SIGGRAPH: Advances in Real-Time Rendering in Games. URL http://www.humus.name/index.php?page=Articles&ID=7; http://s2013.siggraph.org/attendees/courses/session/advances-real-time-rendering-games-part-i. 6, 25, 26, 30

PHARR, M. and HUMPHREYS, G. 2010. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann/Elsevier. ISBN 9780123750792. URL http://www.pbrt.org/. 12, 14

ROGER, D., ASSARSSON, U., and HOLZSCHUCH, N. 2007. Efficient stream reduction on the GPU. In *Workshop on General Purpose Processing on Graphics*

Processing Units. URL http://artis.imag.fr/Publications/2007/RAH07a. 9

SENGUPTA, S., LEFOHN, A. E., and OWENS, J. D. 2006. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, D–26–27. URL http://www.idav.ucdavis.edu/publications/print_pub?pub_id=894. 9

SINTORN, E., KÄMPE, V., OLSSON, O., and ASSARSSON, U. 2014. Per-triangle shadow volumes using a view-sample cluster hierarchy. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 111–118. ISBN 978-1-4503-2717-6. URL http://doi.acm.org/10.1145/2556700.2556716. 11, 26

SUN, B., RAMAMOORTHI, R., NARASIMHAN, S. G., and NAYAR, S. K. 2005. A practical analytic single scattering model for real time rendering. In *ACM SIGGRAPH 2005 Papers*, 1040–1049. URL http://doi.acm.org/10.1145/1186822.1073309. 13, 16

SWOBODA, M. 2009. Deferred lighting and post processing on Playstation 3. Game Developers Conference. URL http://www.technology.scee.net/files/presentations/gdc2009/DeferredLightingandPostProcessingonPS3.ppt. 21

TEBBS, B., NEUMANN, U., EYLES, J., TURK, G., and ELLSWORTH, D. 1989. Parallel architectures and algorithms for real-time synthesis of high quality images using deferred shading. Technical report, UNC Dept. of Computer Science. URL www.dtic.mil/dtic/tr/fulltext/u2/a236590.pdf. 20

THIBIEROZ, N. 2003. Deferred shading with multiple render targets. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*. Wordware Publishing, Incorporated. ISBN 9781556229886. URL http://www.shaderx2.com/; http://books.google.com/books?id=2JJEPQAACAAJ. 20

TOTH, B. and UMENHOFFER, T. 2009. Real-time volumetric lighting in participating media. *EUROGRAPHICS Short Papers*. URL http://sirkan.iit.bme.hu/~szirmay/lightshaft_link.htm. 15

TREBILCO, D. 2009. Light indexed deferred rendering. In *ShaderX7: Advanced Rendering Techniques*. Charles River Media. ISBN 9781584505983. URL http://books.google.com/books?id=iZjrPAAACAAJ. 21

VENCESLAS, B., DIDIER, A., and SYLVAIN, M. 2006. Real time rendering of atmospheric scattering and volumetric shadows. *Journal Of WSCG*, 14(1-3): 65–72. URL http://hdl.handle.net/11025/1362; https://otik.uk.zcu.cz/bitstream/handle/11025/1362/Venceslas.pdf. 15

WHITE, J. and BARRÉ-BRISEBOIS, C. 2011. More performance! Five rendering ideas from Battlefield 3 and Need For Speed: The Run. SIGGRAPH: Advances in Real-Time Rendering in Games. URL http://dice.se/publications/more-performance-five-rendering-ideas-from-battlefield-3-and-need-for-speed-the-run/. 21

WYMAN, C. 2011. Voxelized shadow volumes. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 33–40. ISBN 978-1-4503-0896-0. URL http://doi.acm.org/10.1145/2018323.2018329; http://homepage.cs.uiowa.edu/~cwyman/pubs.html#VoxelizedShadowVolumes. 18

WYMAN, C. and DAI, Z. 2013. Imperfect voxelized shadow volumes. In *Proceedings of the 5th High-Performance Graphics Conference*, 45–52. ISBN 978-1-4503-2135-8. URL http://doi.acm.org/10.1145/2492045.2492050; http://homepage.cs.uiowa.edu/~cwyman/pubs.html#ImperfectVSVs. 30

WYMAN, C. and RAMSEY, S. 2008. Interactive volumetric shadows in participating media with single-scattering. *IEEE Symposium on Interactive Ray Tracing, 2008. RT*, 87–92. URL http://dx.doi.org/10.1109/RT.2008.4634627; http://homepage.cs.uiowa.edu/~cwyman/pubs.html#InteractiveVolumeShadows. 15

Part II
Appended Papers

Paper I

Efficient Stream Compaction on Wide SIMD Many-Core Architectures

Markus Billeter, Ola Olsson, Ulf Assarsson

Reprint from
HPG '09: Proc. of the Conf. on High Performance Graphics
pp 159–166
New Orleans, Louisiana, 2009

10.1145/1572769.1572795
http://link.newq.net/thesis/StreamCompaction
http://link.newq.net/thesis/StreamCompactionPdf

Efficient Stream Compaction on Wide SIMD Many-Core Architectures

Markus Billeter, Ola Olsson, Ulf Assarsson
Chalmers University of Technology

Abstract

Stream compaction is a common parallel primitive used to remove unwanted elements in sparse data. This allows highly parallel algorithms to maintain performance over several processing steps and reduces overall memory usage.

For wide SIMD many-core architectures, we present a novel stream compaction algorithm and explore several variations thereof. Our algorithm is designed to maximize concurrent execution, with minimal use of synchronization. Bandwidth and auxiliary storage requirements are reduced significantly, which allows for substantially better performance.

We have tested our algorithms using CUDA on a PC with an NVIDIA GeForce GTX280 GPU. On this hardware, our reference implementation provides a $3\times$ speedup over previous published algorithms.

Keywords: stream compaction, prefix sum, parallel sorting, GPGPU, CUDA.

Paper II

Real Time Volumetric Shadows using Polygonal Light Volumes

Markus Billeter, Erik Sintorn, Ulf Assarsson

Reprint from
HPG '10: Proc. of the Conf. on High Performance Graphics
pp 39–45
Saarbrucken, Germany, 2010

10.2312/EGGH/HPG10/039-045
http://link.newq.net/thesis/VolumetricShadows
http://link.newq.net/thesis/VolumetricShadowsPdf
http://youtu.be/DUpOr2zdfwE

Real Time Volumetric Shadows using Polygonal Light Volumes

Markus Billeter, Erik Sintorn, Ulf Assarsson
Chalmers University of Technology

Abstract

This paper presents a more efficient way of computing single scattering effects in homogeneous participating media for real-time purposes than the currently popular ray-marching based algorithms. These effects include halos around light sources, volumetric shadows and crepuscular rays. By displacing the vertices of a base mesh with the depths from a standard shadow map, we construct a polygonal mesh that encloses the volume of space that is directly illuminated by a light source. Using this volume we can calculate the airlight contribution for each pixel by considering only points along the eye-ray where shadow-transitions occur. Unlike previous ray-marching methods, our method calculates the exact airlight contribution, with respect to the shadow map resolution, at real time frame rates.

Paper III

Real Time Multiple Scattering using Light Propagation Volumes

Markus Billeter, Erik Sintorn, Ulf Assarsson

Reprint from
I3D '12: Interactive 3D Graphics and Games
pp 119–126
Costa Mesa, CA, USA, 2012

10.1145/2159616.2159636
http://link.newq.net/thesis/MultiScatter
http://link.newq.net/thesis/MultiScatterPdf
http://youtu.be/HtxLVufIHTM

Real Time Multiple Scattering using Light Propagation Volumes

Markus Billeter, Erik Sintorn, Ulf Assarsson
Chalmers University of Technology

Abstract

This paper introduces a new GPU-based, real-time method for rendering volumetric lighting effects produced by scattering in a participating medium. The method includes support for indirect illumination by scattered light, high-quality single-scattered volumetric shadows, and approximate multiple scattered volumetric lighting effects in isotropic and homogeneous media.

The method builds upon an improved propagation scheme for light propagation volumes. This scheme models scattering according to the radiative light transfer equation during propagation. The initial state of the light propagation volumes is based on single-scattered light identified with shadow maps; this allows generation of a high quality initial distribution of radiance. After propagation, the resulting distribution is used as a source of diffuse light during rendering and is also ray marched for volumetric effects from multiple scattering. Volumetric shadows from single-scattered light are rendered separately.

We compare the new method to single-scattered volumetric shadows produced by contemporary techniques, plain light propagation volumes (which this new method extends), and a simple composition thereof.

Keywords: real-time, scattering, light propagation volumes.

Paper IV

Clustered Deferred and Forward Shading

Ola Olsson, **Markus Billeter**, Ulf Assarsson

Reprint from
HPG '12: Proc. of the Conf. on High Performance Graphics
pp 87–96
Paris, France, 2012

10.2312/EGGH/HPG12/087-096
http://link.newq.net/thesis/ClusteredShading
http://link.newq.net/thesis/ClusteredShadingPdf
http://youtu.be/6DyTk7917ZI

Clustered Deferred and Forward Shading

Ola Olsson, Markus Billeter, Ulf Assarsson
Chalmers University of Technology

Abstract

This paper presents and investigates Clustered Shading for deferred and forward rendering. In Clustered Shading, view samples with similar properties (e.g. 3D-position and/or normal) are grouped into clusters. This is comparable to tiled shading, where view samples are grouped into tiles based on 2D-position only. We show that Clustered Shading creates a better mapping of light sources to view samples than tiled shading, resulting in a significant reduction of lighting computations during shading. Additionally, Clustered Shading enables using normal information to perform per-cluster back-face culling of lights, again reducing the number of lighting computations. We also show that Clustered Shading not only outperforms tiled shading in many scenes, but also exhibits better worst case behaviour under tricky conditions (e.g. when looking at high-frequency geometry with large discontinuities in depth). Additionally, Clustered Shading enables real-time scenes with two to three orders of magnitudes more lights than previously feasible (up to around one million light sources).

Paper V

Cloud-Assisted Indirect Illumination on Mobile Devices

Markus Billeter, Lei Yang, Liu Ren, Ulf Assarsson

manuscript - under revision


http://youtu.be/2lPyWkRrVRc

Cloud-Assisted Indirect Illumination on Mobile Devices

Markus Billeter[†], Lei Yang[‡], Liu Ren[‡], Ulf Assarsson[†]
[†]Chalmers University of Technology
[‡]Bosch Research North America

Abstract

In this paper we present a new system design for displaying indirect lighting on mobile devices with support from *the cloud*. One of the major challenges of such systems is to find an efficient and reliable way to partition the indirect lighting computation between server and client. We propose to use Instant Radiosity as the indirect lighting algorithm, and Virtual Point Lights as the intermediate lighting result that is generated on the server and transmitted to the clients. We also propose a variant of the clustered forward shading algorithm to achieve efficient indirect lighting accumulation on low-power mobile clients. Compared with existing solutions, our system requires less bandwidth and is capable of generating view-dependent indirect lighting effects such as glossy reflections. We show that our method scales from low-power mobile clients to powerful PC clients. We also evaluate our system in a few application scenarios, and discuss its scalability with an increased number of clients.

Glossary

AABB – axis aligned bounding box A box whose sides are aligned with the axes of the coordinate system. 3D AABBs can be represented using only six scalar values. 22

absorption The process where a participating medium absorbs light. For rendering purposes, absorption causes light to 'disappear', unlike out-scattering, which 'redirects' light. Extinction is a combination of both absorption and scattering. 12, 14, 62, 64

airlight In-scattered light from single scattering, accumulated along a ray/line segment. See Section 3. 12–17

API – application programming interface An API specifies a set of pre-defined methods and functions that programmers can use to perform specific tasks. 3, 4, 62–65

atomic Used to describe (or refer to) an operation that appears to be uninterruptible. For example, the C/C++ statement `a += b` expands into several primitive operations (e.g., something like load `a`, load `b`, add the two, and store the result back to `a`). Therefore, executing `a += b` two times in parallel, might result in `a` holding the value $(a + b)$[13], and not $(a + b + b)$. If the operation `a += b` were atomic, we would be guaranteed that `a` holds the expected result of $(a+b+b)$. Some atomic operations are supported by the hardware, in which case they typically are accessed through special functions/intrinsics. The GLSL method `imageAtomicOr()` is one such function. 28, 63

cloud A visible mass of liquid droplets suspened in the atmosphere. Sometimes also an invisible mass of computers suspended in data centers. 4, 24

cluster Introduced in Paper **IV**. A cluster is a grouping of view samples based on the view sample's properties. Clusters are a generalization of tiles. We present groupings in 3D (position) and 5D (position and normal); however, other groupings are possible (and might be interesting to explore). 22, 23, 27, 28, 30, 61

cluster key A cluster key identifies a cluster uniquely. A cluster key can be computed from properties associated with a view sample (e.g., the view sample's position). The cluster key serves as an (indirect) index to the cluster's data (e.g., lights assigned to the cluster). Furthermore, some of the cluster's properties, such as the cluster's implicit bounding volume, can be inferred from a cluster key. 23, 27

[13] Or, theoretically, any other value.

color buffer The color buffer stores the color (RGB or RGBA) of each visible fragment/pixel. A color buffer can later be shown to the user on screen, or it can be used for further processing e.g., as a texture. 20, 21, 29, 63, 64

CUDA GPGPU platform developed by NVIDIA, targeting NVIDIA GPUs. CUDA enables development of general-purpose algorithms that run on the GPU without taking the detour via shader-programs. CUDA includes an off-line CUDA C/C++ compiler, that significantly simplifies integration of library methods into GPU programs compared to shader-based approaches. 3, 5, 9, 10, 18, 23, 62, 66

CUDPP The CUDA Data Parallel Primitives Library, an early CUDA library that implements many parallel primitives (see Section 2). The most recent version of the library is available at http://code.google.com/p/cudpp/. 9

deferred shading Deferred shading referes to a mode of rendering. Unlike forward shading, which computes shading of fragments immediatly after generating these, deferred shading *defers* the shading. In deferred shading, the fragment shader instead writes data from each fragment into a G-buffer. Once all geometry has been processed, the view samples that are now stored in the G-buffer are shaded. 20, 62, 63

depth buffer The depth buffer stores the depth (the distance from the camera) of each fragment. The depth buffer is a core component of rendering with rasterization, where it solves the visibility problem (i.e., the problem of hiding occluded surfaces). A fragment's 3D position can be reconstructed from the depth buffer via the fragment's XY-coordinate in the depth buffer, the associated depth stored at that location and the projection matrix used during rendering. 23, 27–29, 63, 65

extinction The combination of absorption and out-scattering. Since both absorption and out-scattering remove light traveling into a certain direction, it is convenient to treat them as a single phenomenon. 12, 13, 61, 64

forward shading The 'traditional' mode of rendering when using APIs like OpenGL and Direct3D. Triangles representing the scene's geometry are transformed and rasterized to the screen. Rasterization generates fragments which are shaded using a fragment shader. The alternative to forward shading is deferred shading. 20, 62

fragment During rasterization, the GPU generates fragments for each pixel/sample that a triangle overlaps with. The fragments are processed by the fragment shader, which determines each fragments resulting properties (e.g., color). Fragments are then merged into the frame buffer, assuming that they pass certain tests such as, for example, the depth test, which determines if the fragment is occluded due to surfaces rendered earlier. 17, 20, 21, 27, 62, 65

frame buffer The buffer to which rendering occurs. A frame buffer has several attachments. These include typically a color buffer and a depth buffer. 17, 22, 28, 29, 62, 63, 65

G-buffer Short for 'geometry buffer'. G-buffers are used in deferred shading. G-buffers store all attributes that are needed to compute shading for each view sample. This includes for example positions, normals, albedo, diffuse and specular colors. G-buffers are typically constructed by rendering geometry with a special fragment shader to a frame buffer with multiple buffer attachments. 20, 21, 23, 62, 65

GLSL The OpenGL Shading Language. OpenGL shaders are written in GLSL. A number of flavours of GLSL exist: different shader types (vertex, fragment, geometry, ...) have access to slightly different functions, and GLSL has seen several revisions of the language with various OpenGL versions. Additional GLSL dialects exist for OpenGL|ES 2.0 and OpenGL|ES 3.0. 16, 17, 28, 61, 63

GPGPU Combination of the terms "general purpose" and "GPU". Refers to the use of GPUs to solve general (non-graphics) problems. 3, 4, 23, 25, 28, 62

GPU – graphics processing unit A specialized processor that traditionally assisted and accelerated rendering tasks. Modern GPUs are massively parallel processors that can be used for both rendering and general purpose computing. GPUs in their various forms are now found in many places, ranging from mobile devices to clusters dedicated for scientific computing. 2–9, 21, 24, 31, 62, 63

imageAtomicOr() GLSL method. Atomic bitwise-`or` operation to OpenGL image resources. Requires OpenGL version 4.2, `EXT_shader_image_load_store` or `ARB_shader_image_load_store`. 28, 61

imageStore() GLSL method. Enables arbitrary writes to OpenGL image resources (`image2D`, `imageBuffer` etc.) from GLSL shaders. Requires OpenGL version 4.2, `EXT_shader_image_load_store` or `ARB_shader_image_load_store`. 27, 28

LPV – light propagation volume A technique where light is injected into a grid and then iteratively transferred to neighbouring cells. Originally used to compute indirect illumination arising from light reflected from one surface to another. Paper **III** adapts LPVs to also consider a participating medium. See Section 3. 18, 19, 30

MSAA Short for Multisample anti-aliasing. MSAA is a anti-aliasing scheme supported by modern GPUs. MSAA reduces the overheads compared to true supersampling by decoupling shading from storage. 26, 27, 29

OpenGL Cross-platform rendering API initially developed by SGI and now managed by the Khronos Group. OpenGL enables the use of GPUs for

rendering tasks. OpenGL comes in frustratingly many forms and versions: the current version is 4.4, but far from all computers support this version. OpenGL|ES 2.0 and OpenGL|ES 3.0 are derived from standard OpenGL and target mobile devices. 3, 4, 16, 18, 23, 29, 62–65

OpenGL|ES 2.0 Rendering API derived from OpenGL that targets mobile devices. Version 2.x was released in 2007 and includes shader programs. It roughly implements functionality comparable to OpenGL version 2.0. 4, 24, 25, 28, 29, 63, 64

OpenGL|ES 3.0 Rendering API derived from OpenGL that targets mobile devices. Version 3.x was released in 2012 and implements many features present in modern-ish OpenGL. 4, 24, 25, 29, 63, 64

participating medium A participating medium is a volumetric effect where a large number of small particles (for example water droplets) affect light passing through the volume. Typical examples of participating media include fog, clouds and smoke. Section 3 is dedicated to the topic of real-time rendering in the presence of participating media. 2, 4, 5, 11, 15, 18, 30, 61, 63, 64

radiance The 5D function $L(\mathbf{x}, \omega)$. Conceptually, the function describes the amount of light traveling into a direction ω at the position $\mathbf{x}$. 14

rasterization The process of translating a vector-based model into a 2D raster image (i.e., a pixel based image). In real-time rendering, models typically consist of triangles; GPUs contain specialized hardware that assists rasterization of triangles. 62

reduction Operation that reduces many inputs into a single result. For example, a reduction with addition as its operator would compute the sum of all input elements. Reductions are further discussed in Section 2. 7–9

RSM – reflective shadow map A shadow map that additionally contains information about reflected light in a color buffer. 18

scan Operation that, for each input element, computes a reduction of all preceding input elements. Scans are further discussed in Section 2. 7–10

scattering The process where a participating medium scatters light, i.e., causes light to change direction. Generally, two flavours of scattering, in- and out-scattering, are considered separately. The former increases the amount of light in a certain direction, and the latter removes light traveling in a given direction. Out-scattering is often combined into extinction together with absorption. 12, 61, 62

shader Short for 'shader program'. Shader programs are (small) program that run at various stages of the graphics pipeline. For example, vertex shaders are run for each input vertex; and fragment shaders are run for each view sample that needs to be shaded. Fragment shaders compute, for

example, the color of a fragment based on interpolated inputs derived from the vertex shader's output. Shader programs in OpenGL are typically compiled on-line – that is, a shader program's source code is passed to the OpenGL API. 16, 17, 27, 29, 62, 63, 65

shadow map A shadow map is a texture that stores the distance (depth) from the light to the surface closest to the light. A point is in shadow if it is further from the light than corresponding point in the shadow map. A shadow map can be created by rendering the scene from the light's point of view to a frame buffer with a single depth buffer attachment. 64

SPU – synergistic processing unit A special purpose processor present in the Cell processor, which is used by the Playstation3. 21

stream compaction Operation that copies selected (*valid*) elements from an input buffer to an output buffer. Copied elements are placed in a compact range in the beginning of the output buffer. Stream compactions are discussed in Section 2 and are the main topic of Paper **I**. 5–10, 65

stream split Operation related to stream compaction. While a stream compaction ignores invalid elements, a stream split places invalid in the second part of the output buffer, after the last valid element. Stream splits are discussed in Section 2. 7–9

texture An array (often, but not necessarily, 2D) that contains data with typically 1-4 channels (e.g., RGB, RGBA). Shaders can sample textures. Sample locations do not need to correspond to exactly one element in the array, but are interpolated according to a pre-determined interpolation mode. (Originally, textures were mainly used to apply images/bitmaps to surfaces.) 13, 28, 29, 62, 65

tile In the context of tiled shading, a tile refers to a 2D rectangle of adjacent pixels or view samples. In this context, tiles typically have an associated depth range derived from the contained view samples. This depth range, together with the 2D extends of the tile, defines a 3D bounding volume in which encompasses all view samples contained in the tile. 21–23, 30, 61, 65

tiled shading Tiled shading reduces shading costs by identifying the light sources that affect each tile. Pixels in each tile then only need to compute lighting from the light sources assigned to their tile. See Section 4 for further information. 21, 22, 26, 28

view ray A ray extending from the viewer (the camera), through a view sample, and into the scene. Figure 6 illustrates two view rays. 12, 13, 15–17

view sample In the context of tiled and clustered shading (Paper **IV**), the term view sample refers to either a fragment generated during forward shading, or a sample retrieved from the G-buffer. The goal of tiled and clustered

shading is to efficiently shade all view samples, regardless of whether forward or deferred shading is employed. 19, 20, 22, 23, 28, 61–65

VPL – virtual point light A light source generated to represent indirect illumination. VPLs are often automatically generated from other light sources. For example, a white light shining at a green surface might generate one or more green VPLs that represent green light reflected of the green surface.. 25

warp In CUDA, a warp is a group of 32-threads that execute in lockstep, i.e., all 32 threads always issued the same instruction (although individual threads can be masked, in which case the instruction has no effect). The hardware schedules execution on a per-warp basis and threads do not migrate between warps. 9, 10