THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Real-Time Shading and Accurate Shadows Using GPGPU Techniques

Ola Olsson

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

Real-Time Shading and Accurate Shadows Using GPGPU Techniques
OLA OLSSON
ISBN 978-91-7385-964-6

©OLA OLSSON, 2014

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 3645
ISSN 0346-718X

Technical Report 104D
Department of Computer Science and Engineering

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

Contact information:
Ola Olsson
Division of Computer Engineering
Department of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden

Phone: +46 (0)31–772 1678
Fax: +46 (0)31–772 3663
Email: ola.olsson@chalmers.se
URL: http://www.cse.chalmers.se/~olaolss

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2014

Real-Time Shading and Accurate Shadows Using GPGPU Techniques

OLA OLSSON
Division of Computer Engineering, Chalmers University of Technology

Abstract

Over the last 10-15 years, computer graphics hardware has evolved at a tremendous pace, with an exponential growth in the number of transistors. This evolution has transformed the Graphics Processing Unit (GPU) from an entirely fixed-function unit into a highly parallel general-purpose architecture that can be programmed using high-level programming languages. During this time, the computational power of the GPU has continually outpaced that of CPUs, resulting in a considerable performance advantage.

As the GPU architecture and capabilities have evolved, so must the algorithms that target these devices. Whereas previously the only road to real-time performance lay in finding ways of mapping problems to the fixed-function hardware, today the bulk of computational power is found in the general-purpose portions of the GPU.

The approach in this thesis utilizes this flexibility and power to explore novel and more efficient algorithms that solve important real-time computer graphics problems by targeting the general-purpose GPU-cores. The thesis focuses on two concrete problem areas: *many-light shading* and *accurate shadows*. For both of these areas, new algorithms are presented that overcome substantial bottlenecks in previous algorithms.

In the area of many-light shading, the thesis presents algorithms that enable a very large number of lights to be shaded, effectively eliminating the number of lights as a primary bottleneck. An efficient method for using virtual shadow maps for hundreds of lights is also introduced, a previously unsolved problem.

For accurate shadows, an algorithm is presented that takes a novel approach to shadow volumes and shows that this results in a robust, flexible, and highly efficient algorithm. The new algorithm is shown to outperform previous work and provide much more predictable performance with changing views.

The underlying problem that is solved in the different algorithms in this thesis is that of intersecting the visible samples with bounding volumes representing light or shadow. The solutions presented in this thesis demonstrate that this can be efficiently achieved using groupings of view samples, combined with a hierarchical acceleration structure. This problem is of a quite general nature, and the solutions derived in this thesis should therefore be applicable to many related real-time rendering problems.

Keywords: rendering, shading, shadows, shadow volumes, transparency, virtual shadow-maps, real time, GPU, GPGPU.

Acknowledgments

My thanks and respect are first and foremost due to Ulf Assarsson, for being a great PhD advisor, with timely feedback and a down to earth approach. I also wish to thank Ulf for all his hard work in establishing the computer graphics research group at Chalmers, and for providing me with the opportunity for these most rewarding six years. I'm very grateful for having been part of this project and allowed to let my ideas run wild on the fields of CG (in a manner of speaking).

Thanks are also due to the rest of the computer graphics group, who are also co-authors on my papers, Erik Sintorn, and Markus Billeter and Viktor Kämpe, for being a great team of people to work with. There are also many other people at Chalmers who have made this a great and stimulating place to conduct my PhD studies, thank you all.

I would also like to thank my Masters Thesis advisor, Angela Wallenburg, and examiner, Reiner Hähnle, for being very supportive in my ambition to become a PhD student, initially by providing a confused undergraduate (that is, me) with a real research challenge for my masters thesis project. This was my first experience with research, and without this and their subsequent encouragement, it is not likely that I would have started down this path.

Big thanks of course to my lovely wife Jaz, who has patiently put up with these years of me delivering weird explanations of computer graphics algorithms and concepts (the last of which involved peas, an ever increasing number of arms, and a keyhole). During my time as a PhD student we also had our two children, Stella and Ruben, and STELLA will be able to read her name right there! Thank you all for just being wonderful, and for introducing a healthy dose of perspective into my life.

I also wish to express my gratitude to remaining family and friends, for always showing interest and at least pretending to consider my absurd interests normal. It is hard to imagine having to go through such an undertaking as this without this acceptance.

I also wish to thank enthusiastic people in the games industry, in particular Emil Persson of Avalanche Studios. Emil has been a fantastic promoter of clustered shading in the industry, and it has been my great pleasure to do presentations together. The people in the Game Technology Brisbane Meetup group have also been inspiring and interested in my work.

Finally, I wish to thank Per Stenström, my examiner, who has always been ready to give constructive feedback. Not least importantly during the writing process leading to this PhD thesis.

List of Appended Papers

This thesis is a summary of the following papers. References to the papers will be made using the Roman numbers associated with the papers.

I Ola Olsson and Ulf Assarsson, " *Tiled Shading*", in *Journal of graphics, GPU, and game tools*, Volume 15, Issue 4, Pages 235–251, 2011

II Ola Olsson, Markus Billeter, and Ulf Assarsson, " *Clustered Deferred and Forward Shading*", in *HPG '12 Proceedings of the Conference on High Performance Graphics*, pp 87–96, June, 2012

III Ola Olsson, Erik Sintorn, Viktor Kämpe, Markus Billeter and Ulf Assarsson, " *Efficient Virtual Shadow Maps for Many Lights*", in *I3D '14: Proceedings of the 2014 symposium on Interactive 3D graphics and games*, to appear, March, 2014

IV Erik Sintorn, Ola Olsson and Ulf Assarsson, " *An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects using per-triangle Shadow Volumes*", in *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2011*, Volume 30, Issue 6, Article No. 153, December 2011

V Erik Sintorn, Viktor Kämpe, Ola Olsson and Ulf Assarsson, " *Per-Triangle Shadow Volumes Using a View-Sample Cluster Hierarchy*", in *I3D '14: Proceedings of the 2014 symposium on Interactive 3D graphics and games*, to appear, March, 2014

Other papers, by the same author, omitted in the thesis:

- Ola Olsson and Ulf Assarsson, " *Improved Ray Hierarchy Alias Free Shadows*", in *Technical Report 2011:09*, Chalmers University of Technology, April 2011
- Markus Billeter, Ola Olsson and Ulf Assarsson , " *Efficient Stream Compaction on Wide SIMD Many-Core Architectures* ", in *Proceedings of the Conference on High Performance Graphics 2009*, August 1-3, 2009

Contents

1 Real-Time Shading and Accurate Shadows Using GPGPU Techniques 1

1 Introduction 1

1.1 Background 2

1.2 Overall Objective and Problem Statements 5

1.3 Main contributions 6

1.4 Thesis Structure 7

2 Real-Time Many-Light Shading 8

2.1 Paper I 8

2.2 Paper II 9

2.3 Paper III 11

3 Real-Time Accurate Shadows 12

3.1 Paper IV 13

3.2 Paper V 14

4 Discussion and Future Work 15

Bibliography 17

2 Paper I: Tiled Shading 21

3 Paper II: Clustered Deferred and Forward Shading 41

4 Paper III: Efficient Virtual Shadow Maps for Many Lights 53

5 Paper IV: An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects using per-triangle Shadow Volumes 63

6 Paper V: Per-Triangle Shadow Volumes Using a View-Sample Cluster Hierarchy 75

Real-Time Shading and Accurate Shadows Using GPGPU Techniques

---

1 Introduction

Computer Graphics (CG) has existed in some form for about as long as there has been computers. In fact, it could be said to have existed even *before* computers, if we consider the output of looms controlled by punched cards. In turn, these machines inspired Charles Babbage into designing the first Turing-complete mechanical computer, the Analytical Engine, in 1837. But let us return to the topic of this thesis, before it turns into a historical novel.

Computer Graphics as a field has been developing since the early 1960s. Since then, it has developed from a niche tool for visualization and engineering into a corner stone for the entertainment industry with practically every released feature film using it for some aspects of production. CG is also an important tool for product visualization. For example, nearly all cars seen in advertising today are virtual. Perhaps most obviously, computer games as we know them could not exist without graphics.

Figure 1.1: *Early graphics programming using punched cards.*

The field of computer graphics can be broadly divided into two main areas: *off-line* algorithms, and *real-time* algorithms. Off-line algorithms may take considerable time, sometimes hours, to produce an image. The goal of these algorithms is to enable movie production and similar, where scene complexity can be significant, image quality requirements are high, and no interactivity is required in the final product. Real-time algorithms, on the other hand, are required to produce a new image quickly, in the order of a few hundredths of a second. The most important goal in this case is to achieve interactivity, as required for games and similar applications. Image quality and scene complexity, while important, are thus secondary concerns and are often traded for performance. This division between off line and real time is not watertight, and considerable overlap exists. Nor is it fixed over time, as hardware capabilities changes, generally enabling algorithms once considered off line to

be used in real-time settings.

Computer graphics *hardware* is a necessary ingredient in nearly all real-time algorithms and has been developing alongside the field since the beginning. In that time, we have seen the transition from specialized, custom-built hardware units, filling entire office buildings, to the situation today, where *graphics processing units* (GPUs) can be found, not only in virtually every desktop and laptop computer, but also in many hand-held devices such as smart phones and tablets. The hardware has also become much more general and flexible, enabling novel algorithms and applications to be executed on the GPUs.

The focus of this thesis is to investigate real-time algorithms that attempt to utilize the potential of modern GPU architectures. The modern GPU, by virtue of expanding processing power and increased programmability, enables new approaches to core real-time graphics problems. Previous solutions have largely been constrained by the need to exploit the limited and fixed-function, but high performance, features of graphics chips. While many real-time algorithms have been proposed over the years, this situation has often favoured the brute-force approach. Now, however, with the rapid evolution of GPU capabilities and supporting programming eco-system, we are at a transition point where more advanced algorithms start to become more effective. This provides the context and common theme for the algorithms in this thesis: they solve important real-time rendering problems using software running on the general-purpose GPU processing cores.

1.1 Background

Real-time graphics, especially in games, has co-evolved with graphics hardware at a break-neck pace for the last decade and a half. Advances in hardware capabilities have enabled new algorithms and higher scene complexity. In turn, this has led to the development of games with increasingly complex visuals, which then fuels demand for better hardware. Driving this cycle is the consumer demand for better performing GPUs, along with the requirements of the work stations needed to produce the ever more complex content for the games and movie industries. Consequently, the performance of GPUs has been increasing exponentially over this time period.

The enabling principle behind this exponential growth is enshrined in the so-called Moore's Law [31]. In short, this law stipulates that transistor counts in microprocessors double every two years, owing to improved manufacturing process technology that shrinks the size of each transistor correspondingly[1]. Figure 1.2 illustrates this trend for around 15 years of CPU and GPU development. CPUs are following the expected curve, but GPUs are on an even steeper trajectory, doubling roughly every 1.5 years. In the logarithmic scale, the difference looks small, but in 2013 a high-end consumer GPU contains four times as many transistors as a corresponding CPU, and the gap is increasing.

An important reason behind this discrepancy is that the typical task a CPU is designed to execute is *sequential*, where each step *depends* on the result of the previous steps.

[1]Interestingly, it has been proposed that the law is a self fulfilling prophecy, and is in fact what motivates the industry to improve process technology [43].

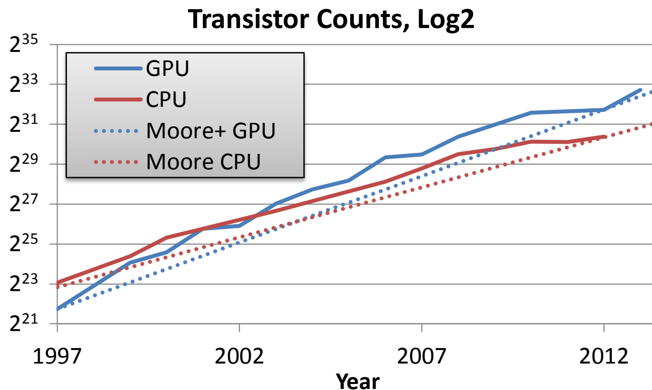


Figure 1.2: *Logarithmic plot of transistor count scaling between 1997 and 2013 for consumer GPUs and CPUs. For each year the data point is the processor with the highest transistor count. Also shown are dashed lines, showing the development if Moore's Law had been strictly adhered to, and chip area had been fixed.*

This is also the prevalent programming model exposed in main-stream programming languages, and as a result, CPU applications struggle to move beyond a single core. Thus, the extra available transistors are not easily translated into improved performance, as increasingly complex logic is needed to attempt to discover independent instructions and execute them in parallel. The typical GPU application has no such problem. For real-time graphics, there is an abundance of *independent* problems ready to be solved. In a modern game, any single frame might draw a few million triangles. This means that a similar number of vertices must be transformed, each of which is independent of the others. Each triangle, in turn, gives rise to a number of independent fragments that require shading calculations. Thus, adding more computational units has a direct impact on performance, as the many independent problems can simply be spread over the larger number of units. Scaling with transistor density is therefore relatively straightforward for GPU architectures, and the effectiveness is high, creating a stronger incentive to expand the number of transistors in each generation. Additionally, as the parallelism is implicitly inherent in the problem, and not explicitly expressed in a program, existing programs (e.g. games) typically benefit directly from advances in hardware. This contrasts with CPU applications that often need considerable re-design to benefit from additional cores.

The consequence, in terms of computational throughput, is illustrated in Figure 1.3. The increase in performance roughly mirrors the growth in transistor count seen earlier, with a large and growing advantage for the GPUs. For the CPU, the bulk of the performance increase is due to increased core counts and wider Single Instruction Multiple Data (SIMD) execution units, i.e. a very similar approach as for GPUs. While sequential programs see little benefit from this development, the techniques and algorithms developed for GPUs are therefore well suited to utilize modern CPU architectures.

Memory bandwidth of a GPU is also much greater than that of a contemporary CPU (in 2013 around six times greater). This is because the typical tasks in graphics can

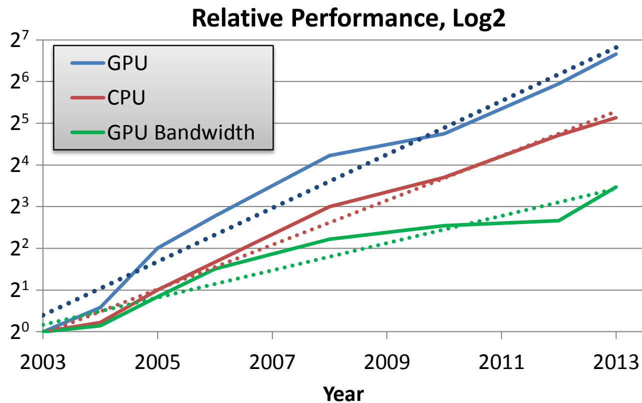


Figure 1.3: *Normalized growth trends of theoretical FLoating-point Operations Per Second (FLOPS), for consumer CPUs and GPUs, and memory bandwidth for GPUs. CPU memory bandwidth shows the same general growth trend as for GPUs, but lower.*

make efficient use of a wide memory interface, which requires large coherent memory transactions. However, the performance increase in this area is much slower than for computational throughput (see Figure 1.3). Like for single-threaded performance, there is no direct scaling effect from smaller process nodes. If anything, smaller chips make it more difficult to place the pins that connect the processor to the memory modules on the circuit board.

However, raw performance is not enough to enable general-purpose computing on the GPU. Another equally important development has been the increase in capabilities and flexibility. The first GPUs were entirely fixed function, which means that they could be configured for a fixed set of options, and were therefore very difficult to use for anything other than graphics. Since that time, they have evolved into something better described as highly parallel general-purpose processors with additional special-purpose, fixed-function hardware units. A modern GPU supports an instruction set that matches that of a modern CPU. Another aspect of this is the memory model, which originally was very restrictive on the GPU. Recent GPUs allow random access to memory and are equipped with a cache hierarchy to speed up unaligned accesses. GPUs also has specific fixed-function support for efficient gather operations, moving non-adjacent data into adjacent threads, in the form of texture units.

This expanding performance advantage is the key motivation for the General Purpose GPU (GPGPU) computing movement. The term GPGPU is used to describe algorithms that perform non-graphics (general purpose) computations on the GPU, but can more generally, as used in this thesis, be used to describe computations that do not require the fixed-function graphics functionality present in the GPU. Early attempts were required to use the graphics APIs to drive computations, which placed quite severe restrictions on the kind of computations that could be performed. This also incurred a substantial overhead, further limiting the achievable gains [34]. With the increased flexibility, GPGPU algorithms can today be more directly implemented using high-level programming languages

such as CUDA C++, which is dialect of the C++ language, with extensions to support parallel execution on the GPU. Several similar efforts exist, for example C++ AMP and OpenCL.

1.2 Overall Objective and Problem Statements

The research in this thesis covers a time where the GPUs are quickly transitioning to fully featured processors with tremendous power. A key aspect is the fact that the proportion of computational power in the general-purpose portions of the GPU are increasingly overshadowing the fixed-function hardware. Figure 1.4 illustrates this by showing a more or less linear scaling for fill-rate or Raster Operations Pipeline (ROP) throughput, which is the rate that pixels can be output to the render target with. In the past, real-time graphics algorithms were required to leverage the fixed-function hardware for the bulk of computations. In practice, this frequently meant problems were reformulated to fit the rasterization pipeline, which offers very high performance but also often made the algorithm less efficient. Today, more efficient algorithms are not only possible, but to support efficient real-time graphics in a future where the fixed-function hardware continues to diminish in importance, new algorithms are required.

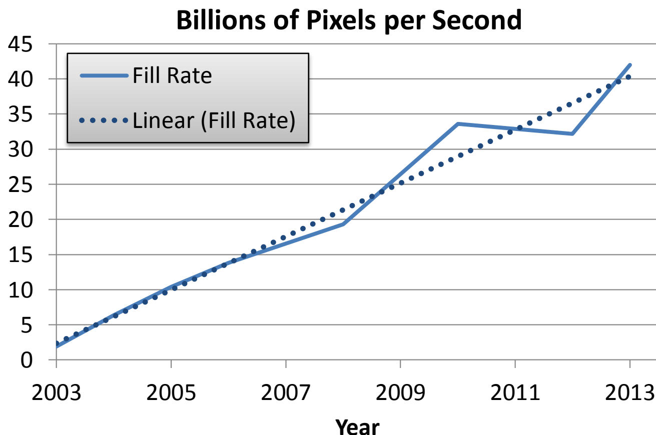


Figure 1.4: *ROP throughput scaling, for high-end NVIDIA GPUs.*

The algorithms and solutions in this thesis are all building on this realization and make use of the general-purpose capabilities of the GPU to solve important problems in real-time graphics in a more efficient manner. The problems examined in this thesis are focused on two separate visual aspects, *shading using many lights* and *accurate shadows*. For both of these problems, there exists previous solutions that are constructed specifically to exploit the fixed-function rasterization hardware. This made them able to deliver real-time performance at a time when the flexibility of GPUs was very limited. However, the same design choice created inherent limits to their efficiency, which are overcome in this thesis.

While using the fixed-function hardware in the wrong way is inefficient, as is shown in this thesis, it excels when used for the specific problem it is designed for. Thus, this thesis does not advocate replacing the entire graphics pipeline with software, only algorithms that can be made more efficient. For example, the fixed-function hardware excels at rasterizing triangles, and does so with very high efficiency. Consequently, if an algorithm requires triangles to be rasterized, it should use this capability. We should not necessarily, however, attempt to map other problems to triangle rasterization.

In this thesis, the term *efficiency* is broadly used when referring to the number of operations performed to achieve some task, and therefore this measure is independent of implementation details. The term *performance*, on the other hand, is used to indicate how fast an implementation completes the task. When discussing efficiency and performance, there are two main bottlenecks that must be addressed: memory bandwidth and computational throughput. Memory bandwidth performance is continually outpaced by advances in computational throughput (Figure 1.3). Accordingly, algorithms that trade some computational efficiency for large improvements in bandwidth are generally preferable.

The first specific problem area is real-time many-light shading. In this application area, lights are generally considered to have a finite range, beyond which the influence is zero. While this fails to model some, especially specular, effects faithfully, it is nevertheless accepted practice in games and similar applications, as it enables a high degree of control over run-time performance. The key problem to be solved is to efficiently establish for each sample, the set of lights affecting it. This is often referred to as *light assignment* and becomes a challenge when there are many lights.

The second problem area is accurate shadows. The previous algorithms offering the highest performance in this area are based on the shadow-volume algorithm [11]. The fundamental idea is to find the polygonal volume enclosing the shadowed regions of the scene and use this to determine what samples are in shadow. The key problem that must be solved in this instance is to determine for each visible sample whether they are within any shadow volume.

In both many-light shading and accurate shadow computation, the problem can be formulated as intersecting the visible samples (the view samples) with some kind of bounding volume. In the case of many-light shading, the volumes are bounding the influence of the light, and for shadow computations, we consider the volumes enclosing the part of the scene that is in shadow. This is the basic computational problem that is solved in the papers encompassed in this thesis.

1.3 Main contributions

This thesis contributes several algorithms and improvements that supplant previous fixed-function oriented algorithms for both many-light shading and accurate shadows, and offer superior real-time performance. The algorithms leverage the general-purpose capabilities of modern GPUs and are highly efficient. In each case, the algorithms overcome a substantial bottleneck in the previous formulation.

At a basic level, this thesis demonstrates that using view-sample groupings is a simple, yet powerful, tool for constructing efficient real-time algorithms. The algorithms in the

thesis all intersect the view-sample groupings with bounding volumes that represent light or shadow, depending on the application. The papers show that this can be approached using Bounding Volume Hierarchies (BVHs), either over the view-samples or volumes that represent light or shadow, and demonstrates that this is possible in real-time for complex scenarios. The presented algorithms apply this to many-light shading and accurate shadows, but the technique has much more general applicability.

Many-Light Shading For many-light shading, this thesis demonstrates several important improvements. The first is the elimination of the memory-bandwidth bottleneck, that is present in previous solutions, and showing that this enables performance scaling with GPU compute capability (Paper **I**). The new techniques are shown to be compatible with *forward shading*, providing a path to many-light shading that avoids the problems associated with deferred shading, for example enabling transparency and Multi Sample Anti Aliasing (MSAA) (Paper **I**). *Clustered Shading* is introduced and shown to enable substantially more robust performance with changing viewing conditions, as well as being capable of supporting many thousands of lights with real-time performance – essentially showing that the *number* of lights need not be a bottleneck (Paper **II**). It is also demonstrated that the efficiency of using clusters is higher than 2D tiles (Paper **II**). The final set of contributions for many-light shading demonstrates how shadows can be efficiently supported, something previously not shown. The main contributions include support for hundreds of shadow-casting lights within a controllable memory footprint, while providing high and uniform quality (Paper **III**).

Accurate Shadows For accurate shadows, this thesis introduces a highly efficient and flexible algorithm for shadow volumes, which utilizes the general-purpose programming capabilities on the GPU. The algorithm represents the first substantially different approach to shadow volumes since the use of the stencil buffer was introduced in 1991 [21], by reconsidering the use of the fixed-function hardware to rasterize shadow quads. The new algorithm provides a robust solution, at the same time as supporting textured and transparent shadow casters, with a small memory footprint (Paper **IV**). The technique is then further improved and shown to consistently outperform traditional stencil-buffer approaches (Paper **V**).

1.4 Thesis Structure

This introduction has provided the background against which this thesis is written and given the overall objective and main contributions. The papers in the thesis focus on two concrete problem areas; many-light shading and accurate shadows. In Section 2, papers **I** to **III**, which focus on many-light shading, are given a more detailed summary of the individual works. Next, in Section 3, the problem of accurate shadows is investigated, and papers **IV** and **V** are summarized. After that, Section 4 concludes the overview part with a discussion of the results and techniques presented in the thesis, along with some thoughts on future work. Finally, there follows a chapter for each of the five papers that represent the research carried out for this thesis.

2 Real-Time Many-Light Shading

Using many lights in real-time applications has been an important goal for many years. The games industry in particular has strived to increase the number of lights to provide enhanced visual quality and realism. Many lights can be used to enable dynamic effects and also to visualize Global Illumination (GI) effects.

Traditional forward shading required a set of lights to be established before each draw call was invoked to render some part of the geometry. This meant that the size and shape of geometry batches directly affects how well the light assignment can be performed. For example, an object with a long extent might be affected by a large number of lights, even though the lights do not overlap, and thus need to be subdivided for efficient shading. This is in direct conflict to the requirements of efficient rasterization, which requires large batches of geometry.

A very successful method for circumventing this problem is deferred shading [41]. Decoupling of geometry and shading is achieved by first rasterizing all geometry into a set of G-Buffers that, for each pixel, store all geometry attributes needed for shading, such as position, color, and shading normal [36]. In a separate pass, all shading can then be performed as operations on the G-Buffers, decoupling shading from the geometry. This frees the rasterizer from needing to compute shading, and batch size can therefore be selected optimally for this stage. The shading stage is also made simpler, as shading for each light and light type can be executed independently. In addition, shading need only be computed for visible samples, in contrast to forward shading where overdraw may discard previously shaded pixels. For a single depth layer, deferred shading is therefore very efficient.

Tebbs et. al. [41] introduced the term deferred shading in 1992, although the technique had been used earlier under different names [13, 36]. With the availability of Multiple Render Targets (MRT) and programmable shaders in commodity GPUs and APIs, such as OpenGL 2.0 and Direct3D 9 [27, 30], the technique became increasingly popular in the real-time community in the early 2000s [42, 16, 19, 37]. The concept of drawing the bounding volumes of the lights as geometry was introduced at this time to ensure the minimal number of samples are shaded. In combination with the use of the stencil buffer to exclude samples outside the volume [4], this yielded a method with optimal efficiency in the number of samples shaded.

The resulting process to perform many-light shading is to first rasterize the geometry to the G-Buffers, and then for each light, rasterize the bounding geometry. The fragment shader will then be executed for each sample covered by the screen-space projection of the light, and can load the geometry attributes from the G-Buffers, compute shading for the light, and add the result in an intermediate buffer.

2.1 Paper I

Problem As noted earlier, GPU compute performance increases much faster than memory bandwidth. The deferred-shading technique, while flexible, very easily becomes bandwidth limited. This is because for each sample that is affected by a light, the shader must load the attributes from the G-Buffer, calculate light contributions from a single

light source, and then add the intermediate result to a buffer. Consequently, while the method is optimal in the *number* of shaded samples, it is limited by the memory-bandwidth performance. In addition, it is very difficult to support transparent geometry with deferred shading, and rendering engines therefore typically resort to a forward shading pass for the transparent geometry and, therefore, needed to maintain a completely separate set of shaders, leading to inconsistent visual results and extra complexity.

In the games development community, a technique based on screen-space tiles has been described to improve deferred shading performance [6, 3, 40, 25, 10]. However, the available material consists of a handful of presentation slides, and little evaluation or comparison is performed.

Methodology In Paper **I**, the starting point is the basic idea of using tiles to solve the many-light problem for deferred shading, called *tiled deferred shading*. The paper examines this technique and attempts to apply it to solve long standing problems in deferred shading. To evaluate the algorithm, a complete implementation using CUDA was created, running all computationally intensive stages on the GPU as a general-purpose processor. The implementation was benchmarked on a set of scenes with several hundred point lights added. Traditional deferred shading was also implemented with the stencil optimization to enable a direct comparison. The paper also introduces a novel generalization, applying the technique tiled shading to forward shading. To establish the efficiency of the algorithms, we measure the number of *lighting computations*. This measure describes the sum of the number of lights affecting each samples.

Contributions Paper **I** examines the use of screen-space tiles to solve the light assignment problem. It is shown that this eliminates the memory bandwidth bottleneck associated with earlier solutions. Paper **I** also introduces a generalization that enables many-light shading with forward shading, called *tiled forward shading*, greatly increasing the flexibility of the technique and making it simple to support transparent geometry and MSAA, which is very difficult with traditional deferred shading. The paper contributes the first detailed description of the technique and a thorough performance analysis, demonstrating that the memory bandwidth is indeed the bottleneck in traditional deferred shading and showing that the performance of tiled deferred shading scales with increased GPU compute capacity.

2.2 Paper II

Problem A fundamental problem with tiled shading is that the two-dimensional screen-space tiles used for light assignment do not match the three-dimensional nature of the scenes it is used to render. This mismatch gives rise to view dependencies that lead to unpredictable performance, which essentially is uncontrollable at content authoring time. In pathological views, the light culling efficiency degenerates to a pure screen-space operation, and even in the case where just a few tiles are affected, the result is poor load balancing in the shading computations. This problem increases when very many lights are used, and in particular when transparent geometry is present. The ability to use

many lights is an important property, as it enables the application to global illumination applications such as photon splatting [39] or manual approximations of such techniques.

Moreover, it is not clear whether using 3D groupings is the best option, given that the scene geometry has several other dimensions that could be explored. For example, the normal direction is implicated in the coherency of lighting calculations. Therefore, it is desirable to explore yet higher-dimensional groupings of view samples.

A problem with this approach is embodied in the so-called *curse of dimensionality*, which simply is a reminder that wherever high-dimensional data is considered, computational cost goes up. The problem at hand is no different, and it is therefore unclear whether the increased cost of many more tiles may in fact be prohibitive.

Methodology To explore this problem space, Paper **II** introduces the concept of *clustered shading*, where the term *cluster* is used to denote a tile with a higher dimensionality than two, in order to differentiate from tiled shading. As the higher dimensionality means that it is not practical to store the full grid, the paper explores several approaches to efficiently determining the actual clusters containing samples. This is achieved through an efficient sorting design, or alternatively using a structure similar to virtual page tables. To solve the light assignment problem, and to support many lights, a light hierarchy was designed, and for each cluster this hierarchy is queried.

The paper also makes use of the fact that since the cluster represents a much smaller sub-division of the viewing volume, the *implied* bounding geometry can be used as a proxy for the samples within, with much greater efficiency than 2D tiles. Therefore the implementation supports both explicit and implicit bounding boxes and bounding cones to explore this trade-off.

The rasterization stage, filling the G-Buffers, is implemented using the fixed-function hardware, as is necessary for real-time performance with complex models. The clustered-shading algorithm stages are implemented on the GPU using CUDA, including light-tree construction and traversal. The implementation makes extensive use of C++ templates to generate efficient specializations of the many different parameters explored.

Contributions The results show that compared to tiled shading, clustered shading achieves higher efficiency and exhibits lower view dependence. The implementation demonstrates that real-time performance is practically achievable, even for very large numbers of lights (up to one million was tested). The overhead for three-dimensional clusters is shown to be low, especially using implicit cluster bounds, even when there are not many lights visible.

The normal-clustering, and the explicit bounds calculations, were found to introduce additional overhead that was not recouped by improved shading performance. This is because the efficiency of the base-line clustered shading with 3D clusters and implicit bounds is already high. However, with improved implementations and GPU hardware, these techniques may well prove valuable in the future, as the efficiency is improved.

Another key result is that clustered forward shading is shown to offer performance that is comparable to the deferred variant. This makes it an important candidate for use with transparency algorithms, for which clusters are especially suited as they perform light

assignment in 3D. Forward shading is also desirable as it enables the use of hardware MSAA, which is very expensive using deferred shading.

2.3 Paper III

Problem In Paper **I** and Paper **II**, and also in previous and concurrent work [6, 3, 40, 25, 10, 18, 17], the effect of adding shadow calculations is not considered. This is generally a result of the ubiquitous trade-off in real-time graphics between performance and visual quality. It is also a highly complex problem area with numerous parameters that will strongly affect performance and capabilities.

Shadows are, however, a highly desirable feature with important implications for visual fidelity and usability. Without shadows, detail in the geometry is lost, in particular when there are many lights. When using automatic or interactive light placement, correct shadows ensure that light does not leak through walls unintentionally.

In the games industry, and real-time graphics community at large, using shadow maps is the de facto standard. Shadow maps suffer from several kinds of aliasing artifacts, but are popular because they can directly leverage the fixed-function hardware rasterizer and offer predictable performance, while being insensitive to the types of geometry used. For this reason, shadow maps need to be evaluated in the context of many lights, if nothing else to provide a base-line benchmark against which to measure more advanced algorithms.

Methodology To design a solution that uses shadow maps for many lights, there are several important concerns. First, effective culling of shadow-casting geometry must be considered. Secondly, the process must be memory efficient, while preferably achieving uniform shadow quality.

The design attempts to meet the first goal by building on clustered shading. The clusters represent a coarse approximation of the visible samples. This information allows the system to establish what regions in each shadow map contain shadow receivers, and therefore also provides a mechanism for detecting the shadow-casting geometry that does not affect these regions.

The second goal is met by designing a system that leverages the recent introduction of *virtual textures* in graphics hardware and APIs. The design in the paper uses virtual-texture support to be able to allocate virtual shadow maps for each light but only commit *physical* storage where needed. The regions where physical backing is required are those containing shadow receivers, as described above. In this way, the design aims to keep the memory usage related to the number of samples needing shadow, while allowing the shadow-map resolution to roughly match the needed resolution.

To evaluate the feasibility of this design, the system was implemented using OpenGL and CUDA. All stages that do not involve rasterization were implemented in CUDA, with sequencing running on the host CPU.

Contributions Paper **III** demonstrates that virtual shadow maps represent a viable path towards real-time many-light shadows. The results show real-time performance for

several hundred high-quality shadows in a complex scene. The memory usage was found to be correlated to the number of shadowed samples, which enables memory budgeting with reasonable constraints. The shadow-culling performance is very high and effective, significantly reducing the number of triangles rasterized into the shadow maps, while incurring little overhead.

The paper also contributes a method for very quickly estimating the required resolution of the shadow map for all lights, in order to achieve uniform quality. This is shown to produce shadows of high visual quality.

Additionally, the paper explores Level of Detail (LOD) schemes, to attempt to establish performance bounds. To this end, GPU ray tracing using a voxel hierarchy is explored. The ray tracing is shown to perform well when there are few samples to shadow for each light, and a hybrid approach combining shadow maps for lights that require many samples, with ray tracing when only few samples are required, is shown to be promising.

Paper **III** presents a high-performing bottom line for real-time many-light shadowing performance, while at the same time providing high quality and efficient use of memory. This makes the paper an important benchmark for future research into this problem space.

3 Real-Time Accurate Shadows

Accurate and robust shadows has been a research topic in computer graphics for a long time. The most successful design to date was introduced by Crow in 1977 [11] under the name *shadow volumes*. In 1991 the modern stencil-buffer based approach was introduced [21]. As the technique makes use of fixed-function hardware available already in early GPUs, the technique quickly became popular, and has been followed by numerous variations and optimizations [7, 8, 22, 15, 26, 1, 9].

The stencil buffer is a full screen buffer that can be set or queried for each generated fragment and can be used to terminate fragments before shading. The basic idea of stencil based shadow volumes is to find the silhouette edges of the shadow-casting object, as seen from the light, and for each of these edges create a *shadow quad*, which extends from the edge to infinity. All shadow quads for an object together with the object itself, precisely bound the volume that is in shadow. The shadow quads are then rasterized to the screen, which must have the depth buffer prepared with the scene geometry. For each front-facing quad, the stencil buffer is incremented by one, and for back-facing quads, it is decremented. Since the stencil operations are performed after the depth test, this will leave all pixels inside the volume with a value greater than zero. These pixels are thus in shadow and can be shaded in a separate shading pass.

There exists competing techniques collectively referred to as alias-free shadow maps, which can also produce accurate shadows efficiently [2, 23]. These techniques use the shadow-mapping algorithm [45] as the starting point and have been realized in real-time implementations utilizing GPGPU techniques [38].

3.1 Paper IV

Problem The various accurate shadow algorithms present all have limitations. Shadow-volume algorithms are able to perform in real-time for many scenes but require geometry to be pre-processed to extract silhouette edges, do not support alpha tested geometry, are not generally robust, and have very high ROP-throughput requirements. There have been numerous papers proposed to advance these issues individually, but they typically sacrifice performance for features or robustness, or generality and scalability for performance.

The use of the fixed-function rasterizer to draw shadow quads leads to very high fill-rate requirements of the shadow-volume algorithms. This is a problem that is strongly view dependent, as it depends on how large the shadow volumes are on screen. Even for moderately complex scenes, it is usually possible to find view points that are several times more costly than other views. This variable and unpredictable performance is problematic for real-time applications, where a maintained smooth and high frame rate is generally desired.

Alias-free shadow maps have fewer problems with robustness and are more flexible. However, existing GPU implementations of techniques suffer from load-balancing problems and resulting unpredictable and poor performance. In addition, memory requirements are quite high.

Methodology Paper **IV** introduces a novel shadow-volume algorithm, based on screen-space tiles, to solves these problems. The algorithm essentially starts from the original idea of shadow volumes and applies that to individual triangles, producing Per Triangle Shadow Volumes (PTSVs) that are rasterized using a hierarchical software rasterizer running on the GPU. This represents a novel path compared to the vast majority of previous work, which focuses on silhouette edge extraction and hardware rasterization.

The algorithm tests each PTSV against a depth hierarchy constructed directly from tiles over the depth buffer, similar to a mip-map. As this is a full hierarchy, no references need to be stored, and the representation is very compact.

By choosing triangle volumes as the primitive, it appears the algorithm would loose a lot of the opportunities for optimization brought by the introduction of the silhouette extraction. This is indeed the case, but it simultaneously enables new, more significant, opportunities. First, by considering a volume primitive, the algorithm can now reject any sample that is outside the volume, whereas stencil-based shadow algorithms need to draw all the shadow quads before this property can be established. The opposite also holds, allowing the algorithm efficiently establish regions that are in shadow. Secondly, where stencil shadows need a counter for each sample, the new method only needs a flag, as a sample can be directly tested for membership in the volume. In addition, the robustness problems are not inherited, and no requirements are made on the scene geometry – any triangle soup can be handled.

To compare efficiency, a measure called *test-and-set operations* is introduced, which describes how many operations are executed. In the case of the stencil based shadows, this corresponds to the number of stencil updates, and for the new method it refers to

the number of nodes visited during the hierarchical rasterization. This measure enables comparing the efficiency of two relatively different approaches.

This algorithm is implemented using CUDA and compared to optimized version of the most prominent versions of shadow volumes, as these represent the best performing real-time accurate shadow methods. The implementation was evaluated on a moderately complex game scene, using an animated camera path.

Contributions Paper **IV** contributes a real-time, robust and very efficient shadow algorithm, demonstrating that even highly tuned algorithms utilizing fixed-function hardware can be outperformed by software running on the general-purpose GPU hardware. The algorithm uses a novel approach to achieve an efficient shadow-volume algorithm based on individual triangle shadow volumes. Compared to traditional shadow volumes, the new approach is much more efficient in the number of test-and-set operations needed to compute shadows. The new approach offers higher performance at high resolutions and is generally shown to exhibit a lower view dependence. The memory overhead is very low, even compared to shadow volumes. The new algorithm can also be trivially extended to support transparent and textured shadow casters, without sacrificing the high efficiency.

3.2 Paper V

Problem The use of screen-space tiles in Paper **IV** means that, while the view dependency of the algorithm is lower than for shadow volumes, it suffers from the same type of view dependency as does the techniques in Paper **I**. This is also a source of inefficiency as, in the presence of depth discontinuities, the higher levels of the depth hierarchy represent very large volumes that intersect many shadow volumes. This also results in poor load balancing in the implementation, as some PTSVs need to visit large portions of the screen to determine if samples intersect.

Methodology To improve these matters, Paper **V** makes use of the cluster concept introduced in Paper **II**, instead of screen-space tiles. The algorithm makes use of small three-dimensional clusters with explicit bounding boxes. From the view samples, a semi-implicit hierarchy of clusters is constructed, making use of the improved atomic operation support found in modern GPUs. The hierarchy is then traversed using the PTSVs as before, at the leaves testing individual samples. These improvements solve the problems with depth discontinuities and therefore also reduce the associated load-balancing problems. However, to completely eliminate these, the algorithm is extended with a two-pass method that performs traversal of all PTSVs to a certain level, and then stores the traversal state to a buffer, before restarting traversal from this depth, with a rebalanced work distribution.

Contributions The resulting algorithm is shown to outperform Paper **IV** and also shadow volumes, consistently. Shadow volumes are demonstrated to offer little performance increase on recent GPU generations, as they scale with ROP throughput. The

new algorithm is able to make effective use of the increased compute capability. The use of clusters and load-balancing measures are shown to be effective and the performance is very robust with respect to changing viewing conditions. The overhead for building the more advanced acceleration structure is modest, even with explicit cluster bounds calculated, and has great potential for many other applications that could benefit from a fast and tight hierarchy over the visible samples. The paper also introduces a new set of culling planes, which significantly improves culling efficiency during traversal, compared to Paper **IV**. The ability to trivially extend to transparent and textured shadow casters is retained and, through the improved load balancing, made more efficient.

4 Discussion and Future Work

This thesis demonstrates that using groupings of view samples provides a useful mechanism for real-time algorithms. Especially, clusters stand out as a useful building block, and the thesis shows several ways in which they can be established very quickly. The applications used to develop these methods are many-light shading and accurate shadows, but there exists many other applications where the fundamental problem is the intersection between view samples and some kind of volume.

In fact, this description applies to much of the real-time research that uses a deferred shading approach to apply effects in screen space. One example is *illumination splatting* to produce GI effects, which has many similarities to many-light shading [39, 12]. Splatting has been implemented using tiled approaches and therefore suffers from the problem of discontinuities that is overcome by using clusters [32, 28]. Another example is *ambient occlusion volumes*, which have the same basic characteristic [29]. Soft shadows using some penumbra-volume representation is another candidate for a clustered approach [5, 14]. These techniques might all be made more efficient by applying the solutions advanced in this thesis.

The many-light shading techniques presented in this thesis will probably become directly useful in the games industry – clustered shading is already implemented in at least one high-profile game [35]. The general techniques in this thesis may also gain attention, with the new generation of high-end consoles having the bulk of their computational resources in GPGPU-capable GPUs. This should lead to an increased interest and pay-off of algorithms that can efficiently exploit this kind of architecture.

Paper **II** and Paper **III** make use of a light hierarchy to quickly establish the lights needed for shading. When the number of lights increases, there emerges an interesting opportunity to using such a hierarchy to merge lights that have a similar influence. This has been explored for off-line rendering and might become applicable in real time in the future [44, 20, 33].

Paper **II** begins the exploration of higher-dimensional clusters. While it is found to not improve performance appreciably for the relatively simple shading application in the paper, the idea remains interesting and potentially useful. In the future, it would be interesting to explore clustering on other attributes and other applications.

Another direction to extend clustered shading in is to consider specular lobes during light assignment [24]. This is a reasonably simple matter of extending the intersection

test between clusters and lights to consider the specular intensity. Doing this would require the normal information in the clusters, as described in Paper **II**. This should increase the visual fidelity by providing specular highlights far from the light sources, but the performance ramifications are unclear.

Volumetric phenomena, such as particles in real-time rendering, are usually bandwidth limited due to high overdraw. These effects might also benefit from the techniques presented in this thesis. However, it is less straight forward as the effects are volumetric and therefore require integration along the entire ray from the eye to the sample. This means that it is not enough to find the intersection with the visible samples. Thus, using a depth hierarchy as in Paper **IV** might be a useful approach, but it seems not unlikely that a solution tailored more towards the specific problem may prove more suitable. The GPU hardware development trends outlined in this thesis show little sign of slowing. Instead, what we find today is that CPUs are becoming more like GPUs, with wide SIMD units and several cores, and are also starting to integrate GPUs on the same chip. This leads to an increasing focus on the use of the GPU for general-purpose programming, to be able to exploit the full potential of the hardware. As the general-purpose compute capabilities of the GPUs increase, yet more algorithms will need to target this hardware to remain on the forefront of real-time performance.

Bibliography

[1] Timo Aila and Tomas Akenine-Möller. A hierarchical shadow volume algorithm. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, HWWS '04, pages 15–23, 2004.

[2] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Proc. of EGSR 2004*, pages 161–166, jun 2004.

[3] Johan Andersson. Parallel graphics in frostbite - current & future. SIGGRAPH Course: Beyond Programmable Shading, 2009.

[4] Jukka Arvo and Timo Aila. Optimized shadow mapping using the stencil buffer. *journal of graphics, gpu, and game tools*, 8(3):23–32, 2003.

[5] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. In *Proc. SIGGRAPH '03*, page 511–520. ACM, 2003.

[6] Christophe Balestra and Pål-Kristian Engstad. The technology of uncharted: Drake's fortune. Game Developers Conference, 2008.

[7] William Bilodeau and Mike Songy. Real time shadows. Creativity 1999, Creative Labs Inc. Sponsored game developer conferences, 1999.

[8] John Carmack. Z-fail shadow volumes. Internet Forum, 2000.

[9] Eric Chan and Frédo Durand. An efficient hybrid shadow rendering algorithm. In *Proc. of the EGSR*, pages 185–195, 2004.

[10] Christina Coffin. Spu-based deferred shading in battlefield 3 for playstation 3. Game Developers Conference, 2011.

[11] Franklin C. Crow. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11:242–248, July 1977.

[12] Carsten Dachsbacher and Marc Stamminger. Splatting indirect illumination. In *Proc. I3D '06*, page 93–100. ACM, 2006.

[13] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22(4):21–30, 1988.

[14] Elmar Eisemann and Xavier Décoret. Visibility sampling on GPU and applications. *Computer Graphics Forum*, 26(3):535–544, 2007.

[15] Cass Everitt and Mark J Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. *arXiv preprint cs/0301002*, 2003.

[16] Rich Geldreich, Matt Pritchard, and John Brooks. Deferred lighting and shading. Game Developers Conference, 2004.

[17] Takahiro Harada. A 2.5D culling for forward+. In *SIGGRAPH Asia 2012 Technical Briefs*, pages 18:1–18:4. ACM, 2012.

[18] Takahiro Harada, Jay McKee, and Jason C. Yang. Forward+: Bringing deferred lighting to the next level. In *Eurographics (Short Papers)*, pages 5–8, 2012.

[19] Shawn Hargreaves and Mark Harris. Deferred shading. NVIDIA Developer Conference: 6800 Leagues Under the Sea, 2004.

[20] Miloš Hašan, Fabio Pellacini, and Kavita Bala. Matrix row-column sampling for the many-light problem. *ACM Trans. Graph.*, 26(3), July 2007.

[21] Tim Heidmann. Real shadows, real time. *Iris Universe*, 18:28–31, 1991. Silicon Graphics, Inc.

[22] Samuel Hornus, Jared Hoberock, Sylvain Lefebvre, and John C. Hart. ZP+: correct Z-pass stencil shadows. In *Proc. I3D '05*, pages 195–202. ACM, 2005.

[23] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4):1462–1482, 2005.

[24] Brian Karis. Tiled light culling. Blog 'Graphic Rants' at http://blogspot.se, 2012.

[25] Andrew Lauritzen. Deferred rendering for current and future rendering pipelines. SIGGRAPH Course: Beyond Programmable Shading, 2010.

[26] Brandon Lloyd, Jeremy Wend, Naga K. Govindaraju, and Dinesh Manocha. Cc shadow volumes. In *EGSR/Eurographics Workshop on Rendering Techniques*, pages 197–206, 2004.

[27] Rob Mace. GL_ATI_draw_buffers. OpenGL Registry, 2002. `http://www.opengl.org/registry/specs/ATI/draw_buffers.txt`.

[28] Michael Mara, David Luebke, and Morgan McGuire. Toward practical real-time photon mapping: efficient GPU density estimation. In *Proc. I3D '13*, I3D '13, page 71–78. ACM, 2013.

[29] Morgan McGuire. Ambient occlusion volumes. In *Proc. of the HPG '10*, page 47–56. Eurographics Association, 2010.

[30] Microsoft. Directx 9.0 features revolutionary high-level shader language. Microsoft Press Release, 2003. `http://www.microsoft.com/presspass/press/2003/jan03/01-22directxhlslpr.mspx`.

[31] Gordon E Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, 1965.

[32] Greg Nichols and Chris Wyman. Interactive indirect illumination using adaptive multiresolution splatting. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):729 –741, 2010.

[33] Jiawei Ou and Fabio Pellacini. LightSlice: matrix slice sampling for the many-lights problem. In *Proc. SA '11*, page 179:1–179:8. ACM, 2011.

[34] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[35] Emil Persson and Ola Olsson. Practical clustered deferred and forward shading. In *Courses: Advances in Real-Time Rendering in Games*, SIGGRAPH '13, page 23:1–23:88. ACM, 2013.

[36] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, 1990.

[37] Oles Shishkovtsov. Deferred shading in S.T.A.L.K.E.R. In *GPU Gems 2*. Addison-Wesley, 2005.

[38] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample-based visibility for soft shadows using alias-free shadow maps. *CG Forum (EGSR 2008)*, 27(4):1285–1292, June 2008.

[39] Wolfgang Stürzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In *Proc. of the Eurographics Workshop on Rendering Techniques '97*, pages 93–102. Springer-Verlag, 1997.

[40] Matt Swoboda. Deferred lighting and post processing on playstation 3. Game Developers Conference, 2009.

[41] Brice Tebbs, Ulrich Neumann, John Eyles, Greg Turk, and David Ellsworth. Parallel architectures and algorithms for real-time synthesis of high quality images using deferred shading. 1992.

[42] Nicolas Thibieroz. Deferred shading with multiple render targets. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*, Wordware game developer's library. Wordware Publishing, Incorporated, 2003.

[43] Video Transcript. Excerpts from a conversation with gordon moore: Moore's law. *Intel Corporation*, 2005.

[44] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. In *Proc. SIGGRAPH '05*, page 1098–1107. ACM, 2005.

[45] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12:270–274, August 1978.

Paper I: Tiled Shading

Ola Olsson and Ulf Assarsson

Abstract: In this article we describe and investigate *Tiled Shading*. The tiled techniques, though simple, enable substantial improvements to both deferred and forward shading. Tiled Shading has previously been talked about only in terms of deferred shading (*Tiled Deferred Shading*). We contribute a more detailed description of the technique, introduce *Tiled Forward Shading* (a generalization of Tiled Deferred Shading to also apply to forward shading), and a thorough performance evaluation.

Tiled Forward Shading has many of the advantages of deferred shading, e.g. scene management and light management is decoupled. At the same time, unlike traditional deferred and tiled deferred shading, *Full Screen Anti Aliasing* and transparency is trivially supported.

We also contribute a thorough comparison of the performance of Tiled Deferred, Tiled Forward and traditional deferred shading. Our evaluation shows that Tiled Deferred Shading has the least variable worst case performance, and scales the best with faster GPUs. Tiled Deferred Shading is especially suitable when there are many light sources. Tiled Forward Shading is shown to be competitive for scenes with fewer lights, while being much simpler than traditional forward shading techniques.

Tiled shading also enables simple transitioning between deferred and forward shading. We demonstrate how this can be used to handle transparent geometry, frequently a problem when using deferred shading.

Demo source code is available online.

Journal of graphics, GPU, and game tools, Volume 15, Issue 4, Pages 235–251, 2011

Vol. 15, No. 4: 235–251

Tiled Shading

Ola Olsson and Ulf Assarsson
Chalmers University of Technology

Abstract. In this article we describe and investigate *tiled shading*. The tiled techniques, though simple, enable substantial improvements to both deferred and forward shading. Tiled Shading has been previously discussed only in terms of deferred shading (*tiled deferred shading*). We contribute a more detailed description of the technique, introduce *tiled forward shading* (a generalization of tiled deferred shading to also apply to forward shading), and a thorough performance evaluation.

Tiled Forward Shading has many of the advantages of deferred shading, for example, scene management and light management are decoupled. At the same time, unlike traditional deferred and tiled deferred shading, *full screen antialiasing* and transparency are trivially supported.

We also present a thorough comparison of the performance of tiled deferred, tiled forward, and traditional deferred shading. Our evaluation shows that tiled deferred shading has the least variable worst-case performance, and scales the best with faster GPUs. Tiled deferred shading is especially suitable when there are many light sources. Tiled forward shading is shown to be competitive for scenes with fewer lights, and is much simpler than traditional forward shading techniques.

Tiled shading also enables simple transitioning between deferred and forward shading. We demonstrate how this can be used to handle transparent geometry, frequently a problem when using deferred shading.

Demo source code is available online at the address provided at the end of this paper.

© Taylor & Francis Group, LLC
ISSN: 2151-237X print/2151-2272 online

1. Introduction

Tiled shading works by bucketing lights into screen-space tiles. Each tile contains a list of (potentially) affecting lights. The tiles can then be processed independently to compute the lighting. We will describe how this technique can be used to great advantage when implementing both deferred and forward shading.

Tiled shading has similarities to *tiled rendering* [Fuchs et al. 89], an old technique wherein the tiling is applied to geometry primitives, instead of lights. Tiled rendering has been unable to scale to the millions of primitives used today. Lights, in contrast, number (at most) in the thousands, for even the most demanding real-time applications. For tiled shading, this enables quick bucketing, and real-time performance.

Tiled deferred shading is not a new technique, but has been described in several recent talks [Balestra and Engstad 08, Andersson 09, Swoboda 09, Lauritzen 10]. In this article, we describe the technique in detail and show how it can be generalized to apply to forward shading as well. We also present an in-depth comparison of the performance, comparing tiled forward shading, tiled deferred shading, and traditional deferred shading. A sample implementation with source code is available online at the address listed at the end of this paper.

1.1. Definitions

Lights, in this article, are *point lights* with a finite *range*, beyond which they do not contribute any lighting. Thus, when we refer to the *light volume*, this is the spherical volume defined by the light position and influence radius. This is obviously not physically correct, but represents a very common type of light in real-time applications. The techniques presented can be applied to arbitrary kinds of lights, but this is beyond the scope of this article.

In the literature, it is not always clear what is meant by *deferred shading*. In this article we refer to the technique whereby all required geometry attributes are rendered into *geometry buffers* (G-buffers [Saito and Takahashi 90]), in a single geometry pass. The G-buffers contain attributes such as position, normal, and material properties for each pixel. This is followed by a lighting pass, during which the lights are applied one at a time by rasterizing the light volumes. Note that this is different from *deferred lighting* [Arvo and Aila 03], which performs light computations only in the deferred pass and adds a separate geometry pass to compute final shading. This technique is also referred to as *light pre-pass rendering* [Engel 09]. Because deferred lighting has the same basic characteristics as deferred shading, we do not evaluate this technique in this article.

We use the term *forward shading* in reference to lighting that is computed in the fragment (or sometimes, vertex) shader as part of the rasterization of the scene geometry. This technique is probably still the most common in real-time applications, such as games.

2. Tiled Deferred Shading

Recently, a technique called *tiled deferred shading* has been discussed in the game development community [Balestra and Engstad 08, Andersson 09, Swoboda 09, Lauritzen 10]. The primary goal is to solve the bandwidth problem, which plagues deferred shading. The problem is that each time a fragment is affected by a light, the geometry information must be read from the G-buffers. Tiled deferred shading also offers other improvements over traditional deferred shading, as listed below:

- G-buffers are read exactly once for each lit fragment.
- Common terms in the rendering equation can be factored out.
- The framebuffer is written exactly once.
- Light accumulation is done in register, at full floating-point precision.
- Fragments (in the same tile) coherently process the same lights.

The first point also benefits computations, as texture fetch and data unpacking need be performed only once. Examples of the common terms that can be factored out are the material diffuse and specular colors, because these are the same for all lights. As can be seen, there are major improvements to both bandwidth and compute. The tiled deferred shading algorithm is summarized in the following steps:

1. Render the (opaque) geometry into the G-buffers.
2. Construct a screen-space grid, covering the framebuffer, with some fixed tile size, $t = (x, y)$, e.g., 32×32 pixels.
3. For each light, find the screen-space extents of the light volume and append the light ID to each affected grid cell.
4. For each fragment in the framebuffer, with location $f = (x, y)$,
    a. sample the G-buffers at f,
    b. accumulate light contributions from all lights in tile at $\lfloor f/t \rfloor$, and
    c. output total light contributions to framebuffer at f.

The first step is an ordinary deferred geometry pass. The second and third steps are part of the light-grid construction process, which can be implemented in several ways (see Sections 6 and 7). The fourth step can be implemented using compute shaders, CUDA, OpenCL, SPUs, or by drawing a full-screen quad, as platform or desire dictates.

2.1. Limitations

Deferred shading has notable weaknesses, some of which are shared by the tiled approach. The most important, perhaps, is when it is combined with *full screen antialiasing* (FSAA). The primary issue is simply the required framebuffer size. At 1080p (1920 × 1080), and with 16 times *multisample antialiasing* (MSAA), a 32-bit color render requires almost 256 Mb to store depth and color samples. For most current graphics hardware, adding several other G-buffers on top of this is simply not possible. Performing the deferred shading post-resolve (i.e., after AA averaging) re-creates the aliasing wherever shading changes quickly. It is, however, possible to compute the shading pre-resolve, and to do so only where edges are present [Swoboda 09, Lauritzen 10] — provided, that is, G-buffers can be made to fit in memory.

Neither deferred shading, nor tiled deferred shading, provides a way to handle transparency. We are left instead with techniques that are approximate [Kircher and Lawrance 09, Enderton et al. 10], or expensive [Everitt 01]. In Section 4, we present a way to conveniently support transparency when using tiled shading.

Another issue, common to both forward shading and tiled deferred shading, is that all lights (that cast shadows) must have their shadow maps built before the shading pass. This is because all lights are in flight simultaneously. Storing shadow maps for hundreds of lights can be prohibitive in terms of memory. In contrast, traditional deferred shading computes all light contributions from each light, one at a time. This means a single shadow map can be reused for all lights.

3. Tiled Forward Shading

Tiled shading is not limited to performing deferred shading. We can also apply the technique to traditional forward shading. We simply access the grid in the fragment shader and apply the relevant lights. This approach has the following advantageous properties:

- Light management is decoupled from geometry.
- Light data can be uploaded to the GPU once per scene.

- FSAA functions as expected.
- Common terms in the rendering equation can be factored out.
- Light accumulation is done in register, at full floating-point precision.
- It performs the same shading function as tiled deferred.

The traditional approach in forward shading is to find and upload a minimal set of lights for each batch of rendered geometry. This is time consuming and imposes an unfortunate conflict between optimal batch sizes and minimizing the number of lights. With the tiled approach, geometry batching can be optimized separately from lighting, and light data can be uploaded once for the entire scene.

Because shading is done in its traditional place in the pipeline, there is no problem applying any FSAA scheme. Also, there are no G-buffers to worry about. Integrating tiled shading into an existing forward shading pipeline is straightforward, owing to the self-contained nature of tiled shading.

The last property is also worth highlighting. Because we use the same data structure for lights, we can use the same shader functions for both deferred and forward shading. This enables much easier transition between the two techniques. We also exploit this to support transparency, as described further in Section 4.

3.1. Limitations

One drawback, compared with deferred shading techniques, is that each fragment can be shaded more than once. When overdraw occurs, the same fragment is influenced by several primitives and consequently shaded for each. This overdraw problem can be addressed by using a pre-z pass, which introduces an extra geometry pass to prime the hierarchical/early Z-buffer. This is already common practice in order to avoid reprocessing complex shaders, and it is ultimately a trade-off between the cost of an extra geometry pass vs. the cost of the redundant shading work. For example, if there are few lights or there is low scene-depth complexity, it might be quicker to skip the pre-z pass.

A related problem occurs when FSAA is enabled. Along primitive edges, fragments can also be shaded several times up to once for each sample. Where there are shading discontinuities, this yields a nicely softened edge. However, if the edge is an interior edge in a continuous mesh, this creates redundant work, whereby each sample is shaded to a very similar (or identical) tone. This is in part a general problem for GPUs, because triangles are becoming smaller (perhaps especially since the introduction of tessellation units). Deferred techniques, in contrast, can analyze the samples in the G-buffers

and compute shading only once, unless a discontinuity is found. In principle, then, deferred shading has an advantage, though efficiently implementing it is not trivial.

4. Transparency

In real-time rendering, transparent geometry is usually handled in a separate pass. The transparent geometry is submitted in (roughly) back-to-front order, with alpha blending enabled. This approach is impossible with deferred shading, because only one layer is represented in the G-buffers. A separate forward pipeline must thus be maintained, complete with light management, which, as mentioned earlier, is both complex and costly—and increasingly so as the number of lights grows.

Using tiled shading, however, we can reuse the grid built for the deferred pass and apply a second tiled forward shading pass with sorting and blending. Because of the fact that all light information is stored in a single global structure (the grid), the rendering pipeline can be mostly the same for both passes. This makes it vastly simpler to support transparency while using deferred shading for the bulk of the geometry.

5. Algorithm Comparison

In Table 1, we summarize the the key differences and properties of the algorithms for easy comparison of important features of the algorithms. Note that the properties are not independent: many depend on other, more fundamental

	Deferred	Tiled Deferred	Tiled Forward
Innermost loop	Pixels	Lights	Lights
Light data access pattern	Sequential	Random	Random
Pixel data access pattern	Random	Sequential	Sequential
Reuse shadow maps	Yes	No	No
Shading pass	Deferred	Deferred	Geometry
G-buffers	Yes	Yes	No
Overdraw of shading	No	No	Yes
Transparency	Difficult	Simple[a]	Simple
Supporting FSAA	Difficult	Difficult	Trivial
Bandwidth usage	High	Low	Low
Light volume intersection	Per Pixel	Per Tile	Per Tile

[a] That is, simple to implement by applying a tiled forward pass reusing the light grid (described in Section 4).

Table 1. Comparison of properties of the algorithms.

features. They are listed in this way, nonetheless, in order to highlight important practical differences.

The most prominent property is the innermost loop structure. If this loop is over the pixels, then lights are accessed in a sequential manner. This makes it possible to reuse shadow maps. Conversely, when the innermost loop is over the lights, the framebuffer data is sequentially accessed. This is what enables fetching the G-buffer data once, providing the large bandwidth savings of tiled deferred shading.

6. Building Tiles

When constructing the grid, our first task is to choose tile size. This choice involves many trade-offs between memory and computation. For example, smaller tiles means more storage and bandwidth use for the grid, but less wasted computation at the light boundaries. It is therefore not likely that there exists a single best tile size for all scene configurations (or even for all views of the same scene).

For this article we use a tile size of 32×32, because this gives three orders of magnitude fewer grid cells than pixels. Consequently, the time and memory spent on managing tiles should not create a bottleneck. We did not experimentally evaluate varying the tile size. However, 16×16 has also been reported to work well [Andersson 09, Lauritzen 10].

6.1. Light Insertion

Next, we must insert the lights into the grid. The simplest way is to find the screen-space extents of the light bounding sphere [Lengyel 02, Sigg et al. 06], and then insert the light into the covered grid cells. This process is simple enough to have a relatively small cost even for hundreds of lights, if implemented on the CPU. For thousands of lights, with high overdraw, it can be implemented on the GPU. The CPU approach is also suitable for older hardware and APIs, which do not support compute shaders.

An interesting way to implement this, which we have not tested, might be to use rasterization to build the grid. This would allow for easy handling of arbitrarily shaped light volumes (e.g., spotlight cones). To ensure that all lights are included, we must use conservative rasterization [Hasselgren et al. 05], because ordinary rasterization samples only fragment centers; and an A-buffer, for example, using per-fragment linked lists [Thibieroz and Grün 10]. It is, however, unclear how well the GPU would perform with these very small render targets (e.g., 60×34, when using tiles of size 32×32 at 1080p).

A quick estimate, obtained by performing standard deferred shading to a render target of this size, indicates that it would be at least around five times slower, compared with the screen-space-bounding-sphere approach.

6.2. Data Structure

To facilitate lookup from shaders, we must store the data structure in a suitable format. We have chosen to use three arrays, as depicted in Figure 1. The *light grid* contains an offset to and the size of the light list for each tile. The *tile light index list* contains light indices, referring to the lights in the *global light list*. This data structure can easily be stored on the GPU as constant buffers or textures. The index list length varies with light overdraw, and can become relatively large. It is thus suitable for storing in a texture.

6.3. Depth Range Optimization

Standard deferred shading implementations often use what is known as the *stencil optimization* [Arvo and Aila 03]. This technique uses an approach analogous to shadow volumes, but with light volumes, to create a stencil mask. The mask lets through only fragments that are actually inside the light volume. This can offer substantial performance improvement where the light is large on screen but affects only a few fragments (for example, a light in the middle of an empty corridor).

Tiled shading can make use of a similar technique, using the depth buffer to compute a min and max Z value for each grid tile. These bounds are then used when adding lights to the grid, to exclude lights that cannot affect the geometry in the tile. Note that tiles that span a depth discontinuity can have

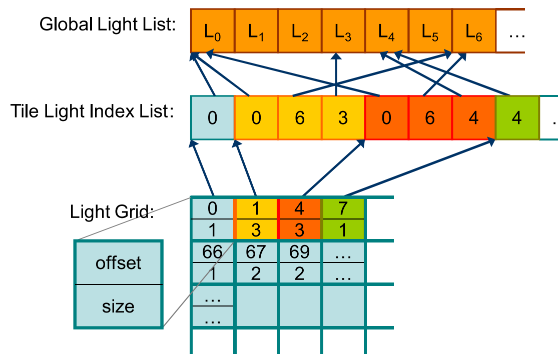


Figure 1. Grid data structure.

a quite large difference between min and max depth. The tiled techniques will therefore always cull somewhat less work than the stencil optimization (this is shown in Figure 3).

Computing the tile min/max depth requires access to the Z-buffer before the grid is constructed. This is not a problem if a pre-z or deferred pass is performed. The min/max operation is suitable for implementation on the GPU, using a parallel reduction per tile. The result can be read back to the host for CPU grid-building, or the grid can be constructed entirely on the GPU. Note that if the CPU is used, the process cannot be completely pipelined with most current APIs.

When rendering transparent geometry (using the tiled forward approach), this geometry is not represented in the depth buffer. We can thus only use the farthest Z value, to reject lights that are hidden by the opaque geometry. We could, if needed, render the transparent geometry to a separate depth buffer with reversed depth test, to find the nearest depth value. This would allow us to reject any light completely in front of all transparent geometry. In scenes where the transparent geometry lies close to the opaque—for example, shallow water—this could be a useful optimization.

7. Single Kernel Tiled Deferred Shading

In past presentations on tiled deferred shading [Andersson 09, Lauritzen 10], the whole process of grid building and lighting application is performed in a single DirectX 11 compute shader. This works by launching one thread group per tile, with one thread per fragment in the tile. Each thread group then independently tests all lights. The six frustum planes for the tile are constructed and then tested against each light bounding sphere. The light list for the tile is built in local, on chip, memory, and the threads involved then switch to lighting the fragments in the tile.

Treating each tile independently leads to a number of redundant calculations. For example, all tiles in a column or row share planes. Also, consider a tile which has at least two opposite neighbors that are affected by a light. This tile must also be affected by the light, without needing any plane tests at all. The approach also requires atomic operations and thread group synchronization, making it unsuitable for older hardware.

An advantage for the single kernel approach is that the process is self-contained and hence simple to implement and integrate. Storing the light index list in shared memory saves some bandwidth but is not a large improvement, because these lists are relatively small.

8. Performance Evaluation

Performance was measured on a PC with an Intel Core 2 Quad at 2.5 GHz, using either an NVIDIA GTX 280 or GTX 480 GPU (as indicated). The

frames were rendered at full HD resolution, 1920 × 1080. The following variations were implemented:

1. **TiledForward** – Tiled forward shading, using the light grid from the pixel shader. Uses the CPU to build the grid, and OpenGL for everything else.
2. **TiledForward-PreZ** – As above, with pre-z pass and depth range optimization (min-max reduction implemented in CUDA).
3. **Deferred** – Standard deferred shading.
4. **Deferred-Stencil** – As above, with stencil optimization to cull unaffected fragments within light volumes.
5. **TiledDeferred** – Tiled deferred shading, using CUDA to build the grid, compute depth range and lighting.

We evaluated several versions of tiled deferred shading, but we report results only for the best performing version. This was an implementation of the process outlined in Sections 2 and 6, using multiple CUDA kernels.

We tried using a full-screen quad in OpenGL, computing the lighting in the fragment shader. However, this approach was substantially slower. For comparison, we also ported the single kernel method used in [Lauritzen 10] to CUDA. We optimized their depth min/max reduction by using a warp-parallel SIMD reduction (similar to `warpReduce` in [Harris 08]) within each warp and only atomic operations between warps. This sped up the reduction by a factor of six for a 16 × 16 tile on our GTX 280. After this optimization, performance is very close to our chosen implementation, with a small advantage for the reported version.

The light model is a fairly ordinary Blinn-Phong model with diffuse and specular reflections. To facilitate this model, the G-buffers are: *Depth*, *Normal*, *Diffuse Color*, *Specular Color and Shininess* and *Emissive and Ambient*.

Each buffer, except depth, stores four 16-bit floating-point values per fragment. However, we also tested using 32 bits, to investigate the impact of G-buffer size on performance (see Tables 2 and 3). The depth buffer always stores a scalar 32-bit floating-point value.

We implemented the 32-bit G-buffers to explore how varying the balance between compute and bandwidth affect the outcome. The expected behavior is that more bandwidth use will favor tiled deferred, whereas increasing compute demand favors traditional deferred. Changing the light model or packing the G-buffers would have a similar impact.

As a test scene we chose the Robots scene from the *Benchmark for Animated Ray Tracing* (BART) suite [Lext et al. 01]. We chose this scene rather than a perhaps better looking game scene because it will allow our experiments

GPU	GTX280		GTX480	
G-Buffer Depth	16-bit	32-bit	16-bit	32-bit
TiledDeferred	15.7	17.2	7.87	10.4
min / max	7.05 / 33.3	8.82 / 34.8	4.19 / 15.9	6.91 / 18.2
TiledFwd	148		43.0	
min / max	26.5 / 410		11.0 / 107	
TiledFwd-PreZ	45.5		30.1	
min / max	12.1 / 125		9.31 / 72.4	
Deferred	38.3	82.1	26.8	51.3
min / max	14.0 / 90.0	27.7 / 197	9.36 / 64.3	18.2 / 121
DeferredStencil	18.4	28.3	12.2	20.2
min / max	8.1 / 39.1	12.4 / 64.8	4.98 / 26.6	9.55 / 43.8

Table 2. Average frame times in milliseconds for the scene with many lights. Min and max frame times are also shown. Results for 16-bit and 32-bit G-buffers are shown, for both the GTX 280 and GTX 480 GPUs. Note that G-buffers are not used for forward shading; thus, only one value is needed for these techniques.

GPU	GTX 280		GTX 480	
G-Buffer Depth	16-bit	32-bit	16-bit	32-bit
TiledDeferred	5.55	7.06	3.34	6.34
min / max	3.00 / 7.13	5.20 / 9.44	2.36 / 4.48	4.68 / 8.98
TiledFwd	9.08		4.03	
min / max	3.39 / 21.1		1.39 / 9.95	
TiledFwd-PreZ	8.39		4.56	
min / max	5.53 / 14.4		3.22 / 9.82	
Deferred	6.55	12.8	4.17	9.36
min / max	3.86 / 10.6	6.65 / 22.7	2.12 / 7.41	5.29 / 15.5
DeferredStencil	6.20	10.6	3.65	8.05
min / max	3.90 / 9.30	6.03 / 16.9	1.65 / 5.81	4.07 / 12.4

Table 3. Average frame times in milliseconds for the scene with few lights. Min and max frame times are also shown. Results for 16-bit and 32-bit G-buffers are shown, for both the GTX 280 and GTX 480 GPUs. Note that G-buffers are not used for forward shading; thus, only one value is needed for these techniques.

to be repeated. The BART suite is freely available, whereas most game data is not.

We augmented the scene with point lights, and created two variations with differing light distributions (for reference, the main street is 29.1 units wide):

- **Many Static Lights** – 924 lights evenly spaced along the animation paths of the robots, with random sideways offsets. Each light has a range of 12.5 units.

- **Few Dynamic Lights** – One light attached to each of the 11 robots. Each light has a range of 40 units.

8.1. Tiled Deferred Shading Performance

Overall, the results confirm that tiled deferred shading is much less variable, with smaller differences between best and worst case performance. This is seen in Figure 2 and Tables 2 and 3.

Traditional deferred shading is usually bandwidth limited. Thus, we expect frame times to scale linearly with G-buffer size. This was confirmed in our experiments, as can be seen in Tables 2 and 3. The tiled deferred variants are much less affected.

Tables 2 and 3 also show results from a GTX 480. This new GPU roughly doubles the (attainable) compute capabilities, whereas bandwidth grows by only about 30%. This also favors the tiled techniques, almost doubling their performance on the new architecture. Traditional deferred shading improves by only about 30%, as expected.

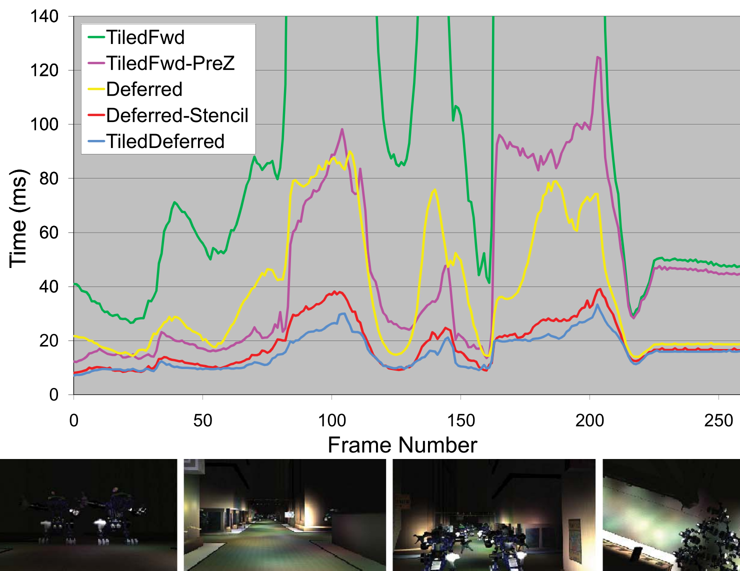


Figure 2. Frame times over the animation sequence in the scene with many static lights, measured on an NVIDIA GTX 280 GPU. The animation is sampled at 5 fps. The peaks correspond to times when the camera is looking through a great many lights. Note that the tiled forward version is clipped, in order to make the presentation clearer. It plateaus at around 400 ms, with a similar shape to the others. The thumbnails below the graph show frames 0, 86, 186, and 259.

Notice that traditional deferred shading with stencil optimization is fairly competitive, when we use 16-bit G-buffers. However, with fatter G-buffers, or increasing compute/bandwidth ratio in the GPU, performance quickly falls behind the tiled techniques. This may not be enough to outweigh the advantages, such as being able to share shadow map storage between lights.

In Figure 3, we show the number of lighting computations (i.e., number of lighting function invocations) performed each frame for the different techniques. As expected, the stencil optimization is the most efficient at culling work, because it is performed at a per-fragment level. TiledFwd is worst, but is substantially improved by the addition of pre-z pass and depth range optimization. Notice the clear similarity to the frame time curves shown in Figure 2. This implies that (perhaps unsurprisingly) there is a reasonably fixed cost per lighting operation, albeit with quite different scales.

8.2. Tiled Forward Shading Performance

Tiled forward shading scales much worse with increasing light overdraw, but the performance curves have the same overall shape (see Figure 2). In fact,

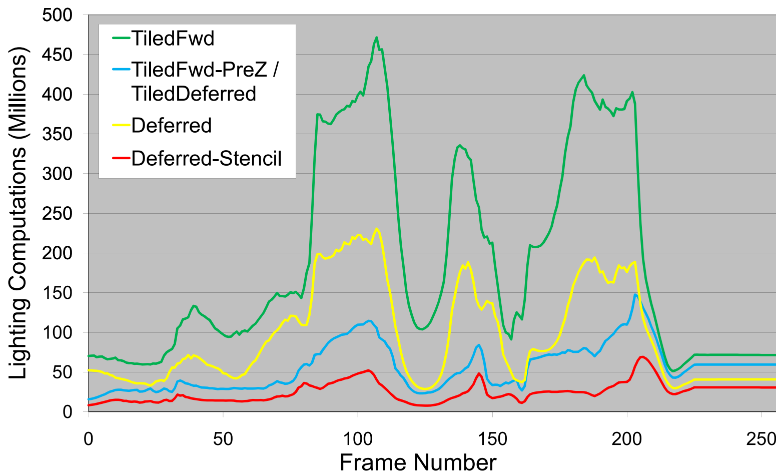


Figure 3. Lighting computations per frame, that is, the number of times the lighting function is executed over the animation sequence in the scene with many static lights. This value corresponds to light volume overdraw for deferred shading, measured using occlusion queries. For TiledFwd-PreZ and TiledDeferred, this is simply the tile size multiplied by the total length of the tile light lists. TiledFwd suffers from geometry overdraw; we measured this using a simple shader, outputting the light counts for each pixel with additive blending enabled.

TiledFwd-PreZ is close to a factor four, and a constant offset, slower than TiledDeferred. Because they ought to perform the same number of lighting computations, the hardware is not utilized as efficiently. One factor could be fragments belonging to different tiles being packed into the same warp (SIMD unit), causing divergence. Also, along the edges of triangles, there can be many pixel quads that are not full, in other words, up to three of the four fragments are outside the triangle (a 2×2 pixel quad is the basic unit handled by fragment shaders), wasting up to 3/4 of computational resources. Furthermore, early Z cull is conservative. Thus, there will be fragments that are shaded, but finally discarded by the Z test, pre-z pass notwithstanding.

On the GTX 480, the TiledFwd technique improves by up to four times for the worst cases. This brings performance closer to expectations, given the high number of lighting computations. The TiledFwd-PreZ shows only modest improvement. It is unclear why it does not improve as much as TiledFwd.

One scenario where tiled forward shading should work well is when scene depth complexity is low, and most light volumes are overlapping the geometry. When lights overlap the geometry, the effect of depth range optimizations is nullified, because these optimizations are designed to cull lights that do *not* overlap the geometry. This description would match a real time strategy (RTS) game pretty well, assuming a top-down view and lots of action on, or near, the terrain.

8.3. Fewer Lights

When rendering fewer lights (i.e., the scene with 11 dynamic lights), the situation is quite different (see Table 3). One important factor is that the total frame time is smaller, and consequently, a larger proportion of time is spent rendering the model into the G-buffers. This favors the forward shading approach, especially on the GTX 480 using 32-bit G-buffers, when compared to the deferred techniques.

However, tiled deferred is still the technique that scales best across platforms and G-buffer depth. It has the fastest worst case performance in all tests performed. The worst case performance is arguably the most important metric for real-time applications, because a stable frame rate is very important for the perceived quality.

At the same time, it is clear that tiled forward shading offers very competitive performance on the GTX 480. Remember that forward shading supports both AA and transparency, and may therefore be a good choice if few lights are used.

8.4. Scaling

To show how the different algorithms respond to varying numbers of lights, we ramped up the number of lights from zero to the maximum number (924) for a fixed view of the scene with many lights. We selected a view that is similar to one of the worst cases in the animation sequence from the previous measurements (see frame 186 in Figure 2). The result is shown in Figure 4.

As can be seen from the figure, the trends are roughly linear for each of the methods. The variability is a result of the lights requiring a varying number of lighting computations, some not being on-screen at all. The relations established by approximately 300 lights continue unaltered for the remainder of the measurement, and are therefore left out of the plot in order to better show the details.

TiledDeferred displays the best scaling with the number of lights, though it also has a higher constant offset. Below about 100 lights, deferred stencil becomes somewhat faster. This reversal is probably tied to light overdraw dropping far enough that the more efficient culling of the stencil optimization outweighs the benefit of reading the G-buffers only once. Where this crossover point occurs, consequently, depends on the relation between compute and bandwidth capabilities (and of course G-buffer size).

Below around ten lights, tiled forward eventually wins out. However, with so few lights, the precise size and placement of each light will influence the results, making results difficult to interpret.

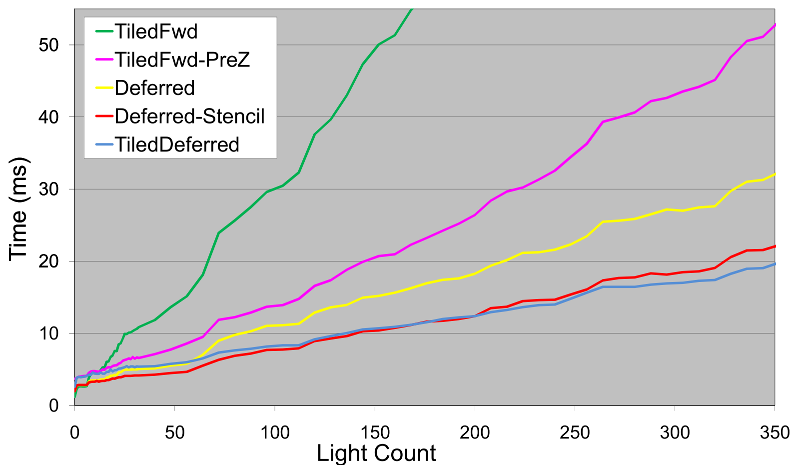


Figure 4. Frame times for an increasing number of lights, measured for a fixed view of the scene with many lights, using an NVIDIA GTX 280 GPU. The discernible trends continue up to the maximum number of lights (924).

References

[Andersson 09] Johan Andersson. "Parallel Graphics in Frostbite - Current & Future." SIGGRAPH Course: Beyond Programmable Shading, 2009. Available at http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf.

[Arvo and Aila 03] Jukka Arvo and Timo Aila. "Optimized Shadow Mapping Using the Stencil Buffer." *journal of graphics, gpu, and game tools* 8:3 (2003), 23–32.

[Balestra and Engstad 08] Christophe Balestra and Pål-Kristian Engstad. "The Technology of Uncharted: Drake's Fortune." Game Developer Conference, 2008. Available at http://www.naughtydog.com/docs/Naughty-Dog-GDC08-UNCHARTED-Tech.pdf.

[Enderton et al. 10] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. "Stochastic Transparency." In *I3D '10: Pro. 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 157–164. New York, NY, USA: ACM, 2010.

[Engel 09] Wolfgang Engel. "The Light Pre-Pass Renderer: Renderer Design for Efficient Support of Multiple Lights." SIGGRAPH Course: Advances in Real-Time Rendering in 3D Graphics and Games, 2009. Available at http://www.bungie.net/News/content.aspx?type=topnews&link=Siggraph˙09.

[Everitt 01] Cass Everitt. "Interactive Order-Independent Transparency." NVIDIA White Paper, 2001. Available at http://developer.nvidia.com/object/Interactive˙Order˙Transparency.html.

[Fuchs et al. 89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. "Pixel-Planes 5: a Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories." In *SIGGRAPH '89: Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 79–88. New York, NY, USA: ACM, 1989.

[Harris 08] Mark Harris. "Optimizing Parallel Reduction in CUDA." NVIDIA CUDA Sample, 2008. Available at http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Optimizing+Parallel+Reduction+in+CUDA#0.

[Hasselgren et al. 05] J Hasselgren, T Akenine-Möller, and L Ohlsson. "Conservative Rasterization." In *GPU Gems 2* edited by M. Pharr and R. Fernando, pp. 677–690. Reading, MA: Addison-Wesley, 2005.

[Kircher and Lawrence 09] Scott Kircher and Alan Lawrance. "Inferred Lighting: Fast Dynamic Lighting and Shadows for Opaque and Translucent Objects." In *Sandbox '09: Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games* New York, NY, USA: ACM (2009), pp. 39–45.

[Lauritzen 10] Andrew Lauritzen. “Deferred Rendering for Current and Future Rendering Pipelines.” SIGGRAPH Course: Beyond Programmable Shading, 2010. Available at http://bps10.idav.ucdavis.edu/talks/12-lauritzen˙Deferred Shading˙BPS˙SIGGRAPH2010.pdf.

[Lengyel 02] Eric Lengyel. “The Mechanics of Robust Stencil Shadows.” Gamasutra, 2002. Available at http://www.gamasutra.com/view/feature/2942/the˙mechanics˙of˙robust˙stencil˙.php.

[Lext et al. 01] Jonas Lext, Ulf Assarsson, and Tomas Möller. “A Benchmark for Animated Ray Tracing.” *IEEE Computer Graphics and Applications* 21 (2001), 22–31.

[Saito and Takahashi 90] Takafumi Saito and Tokiichiro Takahashi. “Comprehensible Rendering of 3-D shapes.” *Proc. SIGGRAPH '90 Computer Graphics* 24:4 (1990), 197–206.

[Sigg et al. 06] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. “GPU-Based Ray Casting of Quadratic Surfaces.” *Proc. Eurographics Symposium on Point-Based Graphics*, 2006.

[Swoboda 09] Matt Swoboda. “Deferred Lighting and Post Processing on PLAYSTATION 3.” *Proc. Game Developer Conference* (2009). Available at http://www.technology.scee.net/files/presentations/gdc2009/DeferredLighting andPostProcessingonPS3.ppt.

[Thibieroz and Grün 10] Nick Thibieroz and Holger Grün. “OIT and GI Using DX11 Linked Lists.” *Proc. Game Developer Conference '10*. Available at http://developer.amd.com/gpu˙assets/OIT%20and%20Indirect%20Illumination%20using%20DX11%20Linkend%20Lists˙forweb.ppsx.

Web Information:

Demo executable with source code is available at http://www.cse.chalmers.se/~olaolss/jgt2011. The demo is implemented using C++ and OpenGL 3.3 with GLSL shaders. Note that this code is different from that which was used for the performance measurements presented in the article. The provided source code is intended to be instructive rather than performing optimally.

Supplementary material can also be found in the publisher's online edition of *journal of graphics, gpu, and game tools*.

Ola Olsson, Chalmers University of Technology Department of Computer Science and Engineering, SE-412 96 Göteborg, Sweden (ola.olsson@chalmers.se)

Ulf Assarsson, Chalmers University of Technology Department of Computer Science and Engineering, SE-412 96 Göteborg, Sweden (uffe@chalmers.se)

Received September 23, 2010; accepted in revised form July 5, 2011.

Paper II: Clustered Deferred and Forward Shading

Ola Olsson, Markus Billeter, and Ulf Assarsson

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).

HPG '12 Proceedings of the Conference on High Performance Graphics, pp 87–96, June, 2012

High Performance Graphics (2012)
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

Clustered Deferred and Forward Shading

Ola Olsson, Markus Billeter, and Ulf Assarsson

Chalmers University of Technology

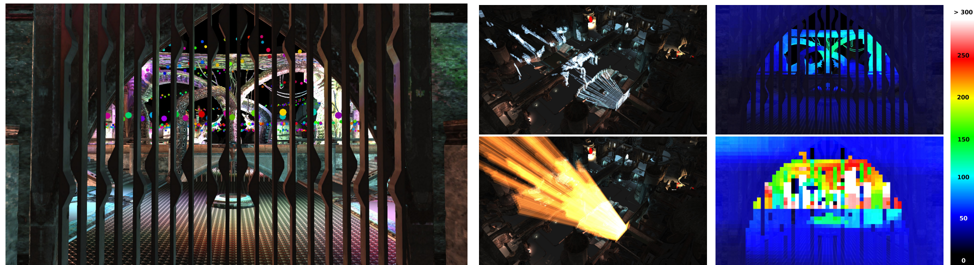


Figure 1: *Clustered Shading groups samples from a view (left image) into clusters (show in blue in the top center image). For shading, each cluster is assigned lights that affect the cluster. Since the clusters are small in comparison to volumes created by e.g. screen space tiling (shown in red in the bottom center image), the number of lighting computations per pixel is kept low (top right image) when compared to Tiled Shading (bottom right image). The colors indicate the number of lighting computations per pixel, ranging from less than* 50 *for blue pixels, to in excess of 300 for white pixels. The scene contains around 2400 light sources, and is rendered in* 17*ms by our method (*2.3*ms for clustering,* 1.5*ms for light assignment and* 5.6 *ms for shading; remaining frame time is dominated by rendering to G-buffers and, here, visualizing light sources with* glutSolidSphere()*), compared to* 26*ms for the Tiled Shading implementation (*1.0*ms for light assignment and* 17.7*ms for shading).*

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).

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

In recent years, *Tiled Shading* [OA11] in various forms has been gathering increased attention in the games development community. The most popular form is *Tiled Deferred Shading*, which has been implemented on both the Sony PlayStation 3 and Microsoft XBox 360 console, as well

© The Eurographics Association 2012.

as PC [BE08, And09, Swo09, Lau10, Cof11]. More recently *Tiled Forward Shading*, has also gained attention [McK12].

Tiled deferred shading removes the bandwidth bottleneck from deferred shading, instead making the technique compute bound. This enables efficient usage of devices with a high compute-to-bandwidth ratio, such as modern consoles and GPUs. Modern high-end games are using tiled deferred shading to allow for thousands of lights, which are required to push the limits of visual fidelity [FC11]. With large numbers of lights, GI effects can be produced that affect dynamic as well as static geometry.

Tiled shading groups samples in rectangular screen-space tiles, using the min and max depth within each tile to define sub frustums. Thus, tiles which contain depth values that are close together, e.g. from a single surface, will be represented with small bounding volumes. However, for tiles where one or more depth discontinuities occur, the depth bounds of the tile must encompass all the empty space between the sample groups (illustrated in Figure 1). This reduces light culling efficiency, in the worst case degenerating to a pure 2D test. This results in a strong dependency between view and performance, which highly is undesirable in real-time applications, as it becomes difficult to guarantee consistent rendering performance at all times.

We introduce *Clustered Shading*, where we explore higher dimensional tiles, which we collectively call *clusters*. Each cluster has a fixed maximum 3D extent, which means that there is no degenerate case depending on the view. Each sample can at worst be over-represented by a fixed volume, and empty space is ignored.

We show how clustered shading can be implemented efficiently on the GPU, supporting both deferred and forward shading implementations. Our implementation shows much less view-dependent performance, and is much faster for some cases that are challenging for tiled shading. We also extend beyond 3D clusters and also use normal information. This is used to implement light *back-face culling* on a per-cluster basis, discarding lights that affect no samples. To robustly support large numbers of lights, we also implement a hierarchical light assignment approach, which is shown to enable real-time performance for up to 1M lights.

2. Previous Work

Deferred shading was first introduced in a hardware design in 1988 [DWS*88], with a more general purpose method using full screen *Geometry Buffers* (G-Buffers) following in 1990 [ST90]. Deferred shading decouples geometry and light processing, making it relatively simple to manage large numbers of light sources. It has become mainstream only in recent years, as hardware has become more powerful and raising the bar on visual fidelity requires more and more lights.

Tiled shading is a relatively recent development that builds upon deferred shading. Aimed primarily at addressing the memory bandwidth bottleneck in deferred shading, it has been implemented in many modern computer games. Since game consoles are highly bandwidth constrained devices, tiled deferred shading has quickly become an important algorithm for high-profile games [BE08, And09, Swo09, Lau10, Cof11, McK12]. The trend for computational power to increase faster than memory bandwidth is also present in consumer GPUs. Tiled shading has been shown to scale well with successive GPU generations [OA11].

2.1. Cluster Determination

To enable efficient processing of clusters, we need some way of determining what clusters are present in a given frame. In a deferred shading setting, this requires analysis of the whole frame buffer, which must be done efficiently on the GPU to minimize data transfers and synchronization. Determining a grouping of samples that goes beyond simple 2D tiling is a fairly common problem in GPU rendering algorithms.

Resolution Matched Shadow Maps (RMSM), must determine which shadow pages are used by the view samples [LSO07]. The method achieves this by first exploiting screen space coherency to reduce duplicate requests from adjacent pixels in screen space. Globally unique requests are then determined by sorting and compacting the remaining requests.

Garanzha et al. [GL10] present a similar technique that they call *Compress-Sort-Decompress* (CSD). Their goal is to find 3D (or 5D) clusters in a frame buffer, which are used to form ray packets. The main differences are that Garanzha et al. treat the frame buffer as a 1D sequence and use *run length encoding* (RLE) to reduce duplicates before sorting. They expand the result after the sorting.

The approaches in both RMSMs and CSD rely on the presence of coherency between *adjacent* input elements, in 2D and 1D respectively. In many cases, this is a reasonable assumption. However, techniques such as *multi sampling anti aliasing* (MSAA) with alpha-to-coverage, or stochastic transparency [ESSL10], invalidate this assumption. Coherency is still present in the frame buffer, but not between adjacent samples. For scenes with low coherence between adjacent samples, both of these methods degenerate to sorting the entire frame buffer.

Virtual textures face a very similar problem as RMSMs, having to determine the used pages in a virtual texture. Mayer [May10] surveys several techniques for solving this problem, all of which are very similar to the above methods. Hollemeersch et al. [HPLdW10] describe a different solution, which instead directly sets a flag in the virtual page table, to indicate that a page is needed. Next the page table is compacted, producing the unique pages needed.

Flagging and compacting page tables does not need to use adjacency to reduce work. All samples requiring the same

© The Eurographics Association 2012.

page will set the same flag, regardless of their position in the frame buffer, eliminating duplicate requests. This method should therefore be more robust with respect to incoherent frame buffers. However, as direct indexing is used, there must be relatively few possible indices (in this case pages).

Liktor and Dashsbacher [LD12] determine and allocate unique shading samples using a related technique. However, because of the high number of unique shading sample identifiers, a direct mapping is not feasible. Also, as they need to allocate space for the samples during the process, they use a more compact hash table to track which samples exist. Space for the samples is allocated in a continuous array, further complicating the process.

3. Clustered Deferred Shading Algorithm

Our algorithm consists of the following basic steps, each of which will be described in more detail in the following sections.

1. Render scene to G-Buffers.
2. Cluster assignment.
3. Find unique clusters.
4. Assign lights to clusters.
5. Shade samples.

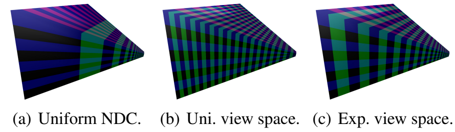
The first step, rendering the model to populate the G-Buffers, does not differ from traditional deferred shading or from tiled deferred shading. The second step computes for each pixel which cluster it belongs to according to its position (and possibly normal). In the third step, we reduce this into a list of unique clusters. The fourth step, assigning lights to clusters, consists of efficiently finding which lights influence which of the unique clusters and produce a list of lights for each cluster. Finally, for each sample, these light lists are accessed to compute the sample's shading.

3.1. Cluster Assignment

The goal of the cluster assignment is to compute an integer *cluster key* for a given view sample from the information available in the G-Buffers. We make use of the position and, optionally, the normal.

There is a potentially limitless number of ways to group view samples. Fundamentally, we desire samples that are close to each other to be grouped together, as they are likely to be affected by the same set of lights. There are many dynamic clustering algorithms available, e.g. k-means clustering, but none of these perform well enough on the millions of samples required to be of interest. Consequently, we employ a regular subdivision, or quantization, of the sample positions, as this is both fast and provides predictable cluster sizes.

The way in which we chose to quantize positions is important in several ways. We desire the clusters to be small, such



(a) Uniform NDC. (b) Uni. view space. (c) Exp. view space.

Figure 2: *Depth subdivision schemes: (a), uniform subdivision of normalized device coordinates; (b), uniform subdivision in view space; and (c), exponential spacing in view space.*

that as few lights as possible affect each, but, conversely, they should contain as many samples as possible to keep the light assignment and shading efficient. We also desire the number of bits required to encode the cluster key to be small and predictable.

A common method is to simply use a world space (virtual) uniform grid [GL10]. This method provides quick cluster key computation, and all clusters are the same size. However, selecting the proper grid cell size requires manual tweaking for each scene, and, depending on scene size, may require a very large number of bits to represent the key. Furthermore, as the grid is viewed under projection, far-away clusters become small on screen. Thus, in large scenes, it is possible to encounter views where many of the clusters are pixel sized, causing poor performance.

We therefore explored an alternative approach, based on the observation that we are only interested in points within the view frustum. Starting with the uniform screen space tiling used in tiled deferred shading, we extend this by also subdividing along the z-axis in view space (or normalized device coordinates), in a manner similar to [HM08]. Viewed in world space, this produces small sub-frustums partitioning the view frustum, see Figure 2.

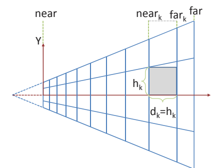


Figure 3: *Exponential spacing in view space. For a given partition k, the near and far planes, as well as cell height and depth are shown.*

The simplest way to perform the z subdivision is to partition the depth range in normalized device coordinates into a set of uniform segments. However, because of the non-linear nature of normalized device coordinates, such a quantization leads to very uneven cluster dimensions. Clusters close to the near plane become very thin, whereas those towards the far

© The Eurographics Association 2012.

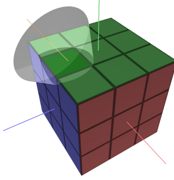


Figure 4: *Quantization of normal directions on the unit cube, using 3×3 subdivisions on each face, and the reconstructed normal cone for one subdivision.*

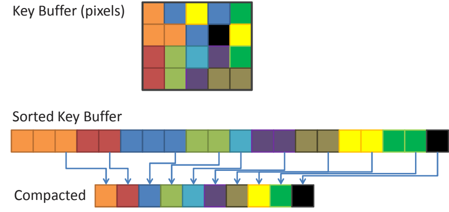


Figure 5: *Sorting and compacting the key buffer to find unique clusters. The cluster keys in the key buffer are sorted and then compacted, to find the list of unique clusters. The sorting is, for instance, based on the view sample's depth and normal direction.*

plane become very long (Figure 2(a)). Uniform subdivision in view space produces the opposite artifact, where clusters near the view point are long and narrow, and those far away are wide and flat (Figure 2(b)).

We therefore choose to perform the subdivision in view space, by spacing the divisions exponentially to achieve self-similar subdivisions [LTYM06], such that the clusters become as cubical as possible (Figures 2(c) and 3).

In Figure 3, we illustrate the subdivisions of a frustum. The number of subdivisions in the Y direction (S_y) is given in screen space (e.g. to form tiles of 32×32 pixels). The near plane for a division k, $near_k$, can be calculated from

$$near_k = near_{k-1} + h_{k-1}.$$

For the first subdivision, $near_0 = near$, i.e. the near viewing plane. For a given field of view of 2θ, we find that

$$h_0 = \frac{2\,near\,\tan\theta}{S_y}.$$

It follows that $near_k$ can be computed using the following expression:

$$near_k = near\left(1 + \frac{2\tan\theta}{S_y}\right)^k. \quad (1)$$

Solving Equation 1 for k, we find that

$$k = \left\lfloor \frac{\log\left(-z_{vs}/near\right)}{\log\left(1 + \frac{2\tan\theta}{S_y}\right)} \right\rfloor. \quad (2)$$

Using Equation 2, we can now compute the cluster key tuple (i, j, k) from screen-space coordinates (x_{ss}, y_{ss}) and the view-space depth z_{vs}. Coordinates (i, j) are the screen space tile indices, i.e. for tile size (t_x, t_y), $(i, j) = (\lfloor x_{ss}/t_x \rfloor, \lfloor y_{ss}/t_y \rfloor)$.

Using our more dynamic definition of a cluster opens up for the ability to use attributes other than the position to define the cluster key. We extend the cluster key with a number of bits that encode a quantized normal direction (Figure 6). We quantize normals by cube face and a discrete 2D grid over each face, as illustrated in Figure 4. Clustering on normals improves culling of lights (see Section 3.3).

3.2. Finding Unique Clusters

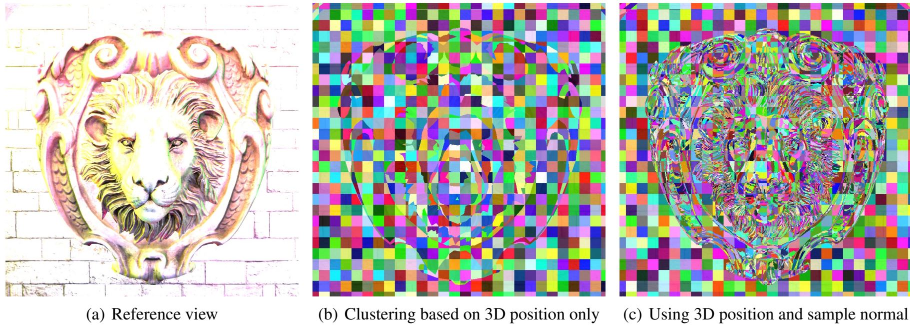
We will here present two different options that we use for identifying unique clusters: with sorting and with page tables.

The perhaps most obvious method to find the unique clusters in parallel is to simply sort the cluster keys, and then perform a compaction step that removes any with an identical neighbour (see Figure 5). Both sorting and compaction are relatively efficient and readily available GPU building blocks [HB10, BOA09]. However, despite steady progress, sorting remains an expensive operation, and we therefore explore better performing alternatives.

As noted in Section 2, methods that rely on *adjacent* screen-space coherency are not robust, especially with respect to stochastic frame buffers. We therefore focus on techniques that do not suffer from this weakness.

3.2.0.1. Local Sorting In our first technique, we sort samples in each screen space tile locally. This allows us to perform the sorting operation in on-chip shared memory, and use local (and therefore smaller) indices to link back to the source pixel. We extract unique clusters from each tile using a parallel compaction. From this, we get the globally unique list of clusters. During the compaction, we also compute and store a link from each sample to its associated cluster.

3.2.0.2. Page Tables The second technique is similar to the page table approach used by virtual textures (Section 2). However, as the range of possible cluster keys is very large, we cannot use a *direct* mapping between cluster key and physical storage location for the cluster data; it simply would typically not fit into GPU memory. Instead we use a *virtual* mapping, and allocate physical pages where any actual keys needs storage. Lefohn et.al. [LSK*06] provide details on software GPU implementation of virtual address translation. We exploit the fact that all physical pages are allocated in a compact range, and we can therefore compact that range to find the unique clusters.



(a) Reference view (b) Clustering based on 3D position only (c) Using 3D position and sample normal

Figure 6: *Results of different clustering methods. (a) The rendered and lit reference view is shown to the right. (b) The center image shows the results of clustering on position only. (Each cluster is assigned a random color.) (c) Clustering based on position and normals is shown to the right . In both cases, flat regions produce a clustering very similar to screen space tiling.*

Whether using sorting or page tables, the cluster key defines implicit 3D bounds and, optionally, an implicit normal cone. However, as the actual view-sample positions and normals typically have tighter bounds, we also evaluate explicit 3D bounds and normal cones. We compute the explicit bounds by performing a reduction over the samples in each cluster (e.g., we perform a min-max reduction to find the AABB enclosing each cluster). The results of the reduction are stored separately in memory.

When using page tables, the reduction is difficult to implement efficiently. Because of the many-to-one mapping from view samples to cluster data, we would need to make use of atomic operations, and get a high rate of collisions. We deemed this to be impractically expensive. We therefore only implement explicit bounds for the first technique based on sorting (after the local sort, information about which samples belong to a given cluster is readily available).

3.3. Light Assignment

The goal of the light assignment stage is to calculate the list of lights influencing each cluster. Previous designs for tiled deferred shading implementations have by and large utilized a brute force approach to finding the intersection between lights and tiles. That is, light-cluster overlaps were found by, for each tile, iterating over all lights in the scene and testing bounding volumes. This is tolerable for reasonably low numbers of lights and clusters.

To robustly support large numbers of lights and a dynamically varying number of clusters, we use a fully hierarchical approach based on a spatial tree over the lights. Each frame, we construct a *bounding volume hierarchy* (BVH) by first sorting the lights according to the Z-order (Morton Code) based on the discretized centre position of each light. We derive the discretization from a dynamically computed bounding volume around all lights.

The leaves of the search tree we get directly from the sorted data. Next, 32 consecutive leaves are grouped into a bounding volume (AABB) to form the first level above the leaves. The next level is constructed by again combining 32 consecutive elements. We continue until a single root element remains.

For each cluster, we traverse this BVH using depth-first traversal. At each level, the bounding box of the cluster (either explicitly computed from the cluster's contents or implicitly derived from the cluster's key) is tested against the bounding volumes of the child nodes. For the leaf nodes, the sphere bounding the light source is used; other nodes store an AABB enclosing the node. The branching factor of 32 allows efficient SIMD-traversal on the GPU and keeps the search tree relatively shallow (up to 5 levels), which is used to avoid expensive recursion (the branching factor should be adjusted depending on the GPU used, the factor of 32 is convenient on current NVIDIA GPUs).

If a normal cone is available for a cluster, we use this cone to further reject lights that will not affect any samples in the cluster. This happens if ω, the angle between the incoming light direction from the centre of the cluster AABB (d_i) and the normal cone axis (a), is greater than $\pi/2 + \alpha + \delta$. The angle α is the normal-cone half angle, and δ is the half angle of the cone from the light enclosing the cluster AABB (see Figure 7).

3.4. Shading

Shading differs from Tiled Shading only in how we look up the cluster for the view sample in question. For Tiled Shading, a simple 2D lookup, based on the screen-space coordi-

© The Eurographics Association 2012.

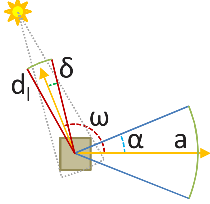


Figure 7: *Back-face culling of lights against clusters. A normal cone (blue), with the opening angle α is derived from or stored with the cluster. The normals of the samples contained in this cluster are all within this cone. The cone originating at the light source and enclosing the cluster (dashed grey – geometrically equivalent to the red cone) gives the angle δ. If the angle between the incoming light and the axis of the normal cone (ω) is greater than $\pi/2+\alpha+\delta$, the light faces the back of all samples in the cluster, and can therefore be ignored.*

nates, is sufficient to retrieve light-list offset and count. However, for clustered approaches, there no longer exists a direct mapping between the cluster key and the index into the list of unique clusters.

In the sorting approach, we explicitly store this index for each pixel. This is achieved by tracking references back to the originating pixel, and, when the unique cluster list is established, storing the index to the correct pixel in a full screen buffer.

When using page tables, after the unique clusters are found, we store the cluster index back to the physical memory location used to store the cluster key earlier (using the same page table as before). This means that a virtual lookup for the cluster key will yield the cluster index. Thus, each sample can look up the cluster index using the cluster key computed earlier (or re-computed).

4. Implementation and Evaluation

We implemented several variants of the new algorithm using OpenGL and CUDA. The variants are as follows (suffixes used are documented in Table 1):

- *ClusteredDeferred[Nk][En][Eb][Pt]* – clustered deferred shading.
- *ClusteredForward* – clustered forward shading. Clustered forward shading requires a pre-z pass to prime the depth buffer, which is used for clustering. Currently only implemented with page tables.

Additionally, we implemented the following methods for comparison, as described in [OA11]:

- *Deferred*, traditional deferred shading, with stencil optimization. This means that light assignment will be exact per sample using a stencil test [AA03].
- *TiledDeferred*, standard tiled deferred shading.
- *TiledDeferredEn*, tiled deferred shading with explicit normal cones computed per tile.
- *TiledForward*, standard tiled forward shading, with a depth pre-pass, to enable min-max culling of lights.

Table 1: *Suffixes identifying variations of the clustered methods.*

Suffix	*Meaning*
Nk[*X*]	Clustering based on normal, using $X \times X$ subdivisions to a cube face.
En	Explicit normals cones are derived and used.
Eb	Explicit Bounds (3D AABB) are derived and used.
Pt	Page Tables are used to find the unique clusters

4.1. Cluster Key Packing

For maximum performance when using sorting or page tables, we wish to pack the cluster key into as few bits as possible. We allocate 8 bits to each i and j components, which identify the screen-space tile the cluster belongs to. This allows up to 8192×8192 size render targets (assuming screen-space tile size of 32×32 pixels). The depth index k is determined from settings for the near and far planes and Equation 2. In our scenes, we found 10 bits to be sufficient. This leaves up to 6 bits for the optional normal clustering. Using 6 bits, we can for instance support a resolution up to 3×3 subdivisions on each cube face ($3 \times 3 \times 6 = 54$ and $\lceil \log_2 54 \rceil = 6$). For more restricted environments, the data could be packed more aggressively, saving both time and space.

4.2. Tile Sorting

To the cluster key (between 10 and 16 bits wide) we attach an additional 10 bits of meta-data, which identifies the sample's original position relative to its tile. We then perform a tile-local sort of the cluster keys and the associated meta-data. The sort only considers the up-to 16 bits of the cluster key; the meta-data is used as a link back to the original sample after sorting. In each tile, we count the number of unique cluster keys. Using a prefix operation over the counts from each tile, we find the total number of unique cluster keys and assign each cluster a unique ID in the range [0...numClusters). We write the unique ID back to each pixel that is a member of the cluster. The unique ID also serves as an offset in memory to where the cluster's data is stored.

Bounding volumes (AABB and normal cone) can be reconstructed from the cluster keys, in which case each cluster

© The Eurographics Association 2012.

Figure 8: *A view of the Crytek Sponza scene, with 10k lights randomly placed. The tree branches cause discontinuities in the depth buffer, making it more challenging for tiled deferred shading.*

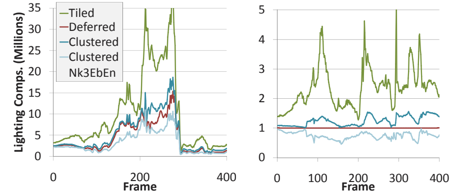


Figure 9: *(left) Millions of lighting computations performed along a fly-through of the Necropolis scene. (right) Same data, normalized to the Deferred method.*

only needs to store its cluster key. For explicit bounding volumes, we additionally store the AABB and/or normal cone. The explicit bounding volumes are computed using a reduction operation: for instance, AABBs can be found using a min- and a max-reduction operation on the sample positions. The meta-data from the locally sorted cluster keys gives us information on which samples belong to a given cluster.

4.3. Page Tables

We implemented a single level page table using a two pass approach. First the required pages are flagged in the table. Then, the physical pages are allocated using a parallel prefix sum, and finally the keys are stored into the physical pages. Performing the physical page allocation on the fly in a single pass was more than 2 times slower, but could still be viable on hardware with faster atomic operations.

4.4. Light Assignment

As described in Section 3.3, we construct a search tree over the lights each frame. Construction relies on efficient sorting functions; here we use the sorting function provided by Thrust [HB10]. To construct the upper levels of the tree, we launch a CUDA warp (32 threads) for each node to be constructed. The warp performs an in-warp parallel reduction over the children's bounding volumes.

For traversal, we again take advantage of the 32-wide fanout of the search tree. For each cluster we allocated a warp that traverses the tree in depth-first order. Each thread in the warp tests the 32 bounding volumes of the children in parallel. By providing unrolled implementations for trees of depth up to 5, we can avoid expensive recursion in CUDA. With a depth of 5, we can support up to 32 million lights, which we deemed to be sufficient (it is trivial to expand this).

5. Results and Discussion

We measured performance for the algorithm and variants described in the previous section, and measurements are performed on an NVIDIA GTX 480 GPU, unless otherwise indicated. We used the set of scenes listed below.

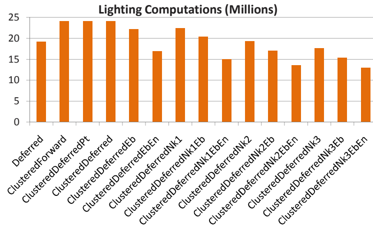
- *Necropolis*. Scene from the Unreal Development Kit [Epi11] (Figure 1). The scene contains 653 lights, with bounded ranges. The majority are spot lights. However we treat all lights as point lights (this is a limitations in our implementation). The scene contains around 2M triangles and is normal mapped. We created a camera animation covering the length of the map (see supplemental video). To bring the number of lights up further, we added several cannon towers to the scene which shoot out colourful spheres, bringing the total number of lights up to around 2500 during the animation.
- *Sponza*. We used the version of sponza made available by Crytek [Cry10] (Figure 8). To make the scene more challenging, with more discontinuities, we injected a set of bare trees. We generated 10k random lights within the scene AABB.

5.1. Performance Analysis

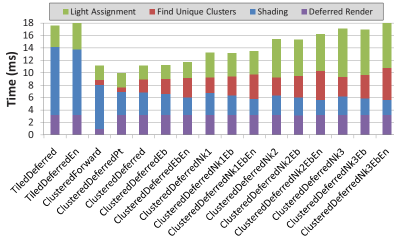
The main advantage of clustered shading over tiled shading is the reduced view dependence. By avoiding empty space, efficiency should be similar to that of deferred shading with stencil optimization and less variable than tiled shading. This is shown in Figures 9 and 10(a), which both adopt the lighting computations metric from [OA11].

Since clustering and light assignment introduce overheads, it is expected that tiled shading performs better when there are fewer lights, or few discontinuities. Clustered shading is still expected to have less view-dependent variability in frame times. Figure 11 confirms that this is the case for the necropolis scene, which has relatively few discontinuities and lights. Even the most complex clustered algorithm tested (ClusteredNk3EbEn), offers worst case performance comparable to tiled deferred. This is also the case for the more challenging scene shown in Figure 10(b), with many discontinuities and lights, indicating greater robustness for clustered shading. We also see that the best performing clustered variant (ClusteredDeferredPt) is around 50% faster in the worst case on the necropolis animation.

© The Eurographics Association 2012.



(a) Efficiency, millions of lighting computations.



(b) Performance, milliseconds for important stages.

Figure 10: *Performance measured for the tested algorithms for the view of the crytek sponza scene shown in Figure 8. Tiled variants have been excluded from (a), as they make comparison difficult. They perform around 90 million lighting computations. For the same reason, Deferred and TiledForward have been excluded from (b). Deferred takes a total of* 97.1 *ms, and TiledForward* 23.6 *ms to render.*

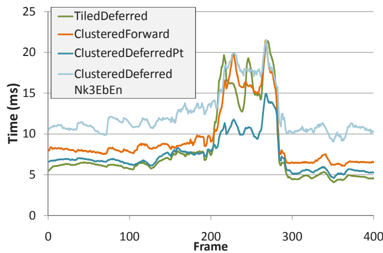


Figure 11: *Run time performance of some the algorithm variants over the Necropolis scene animation.*

ClusteredForward offers very competitive performance, similar to TiledDeferred in the Necropolis animation sequence (Figure 11). This is interesting as, by using forward shading, this variant inherently support MSAA, custom material shaders, and sidesteps the issue of G-Buffer storage. This is remarkable since TiledForward performs significantly worse than TiledDeferred (which is why TiledForward was excluded from Figure 11).

Table 2: *Light assignment performance scaling with an increasing number of randomly distributed lights.*

#lights	Clustered Light Assignment Time	Tiled Light Assignment Time
32	0.71 ms	0.24 ms
1024	0.73 ms	0.51 ms
32768	1.42 ms	9.31 ms
1048576	5.73 ms	341.56 ms

Run-time performance is influenced by many factors, including the number of lights, light density, the level of discontinuities, algorithm complexity, and various implementation details. In Figure 12, we explore the first three of these options. While the crossover point between tiled and clustered implementations is at most around $2k$ lights, the most important conclusion is that clustered shading is very competitive even for cases with very few lights.

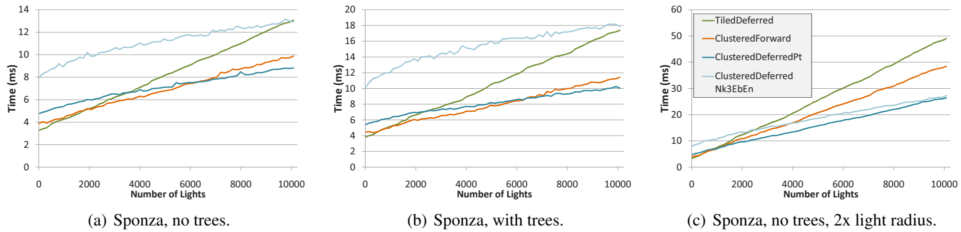
Using normal cones and explicit bounds improves efficiency and *shading time* in all methods tested (Figures 9 and 10). However, as other stages become slower, this does not translate into faster rendering overall. Even the relatively modest overhead of adding normal cone construction to tiled deferred (TiledDeferredEn) is too large to offer any net benefit. This affirms that the major performance gain comes from the move beyond 2D tiles. To make these more advanced clusterings attractive, either faster methods for light assignment and clustering must be found, or the shading cost must increase.

As our clustered shading implementation uses a light hierarchy for light assignment, it should scale well with increasing numbers of lights. Table 2 shows this, where we compare the hierarchical light assignment against the brute-force approach used by the tiled implementation. For small numbers of lights, various overheads dominate the assignment time, making the clustered variant slightly more expensive. At 1M lights, our clustered-shading implementation runs at over 35 fps, where the lights are uniformly distributed and up to 100 lights ($\sim$ 45 on average) end up influencing each cluster.

6. Conclusion and Future Work

In this paper, we have presented and evaluated *Clustered Shading*. In clustered shading, we group similar view samples according to their position and, optionally, normal into clusters. We then determine what light sources potentially influence what clusters. Compared to tiled shading, clusters generally are smaller, and therefore will be affected by fewer light sources. The optional per-cluster normal-information allows us to cull back-facing light sources against clustering, further reducing the number of light sources affecting each

© The Eurographics Association 2012.



(a) Sponza, no trees. (b) Sponza, with trees. (c) Sponza, no trees, 2x light radius.

Figure 12: *Crossover points for various algorithms and numbers of lights for the view of Sponza seen in Figure 8. Note that (a) and (c) use the same view, but without trees, and therefore contain fewer discontinuities.*

cluster. We have shown that efficiency is indeed superior, and that performance is more robust with respect to changing viewing conditions. Our implementation shows that both clustered deferred and forward shading offer real-time performance and can scale up to 1M lights. In addition, overhead for the clustering is low, making it competitive even for few lights.

In the future, we would like to explore approximative lighting, where a heuristic is used to determine if all view samples in a cluster are affected approximately equally by a certain light. If so, the lighting for that light source is evaluated once and re-used for all samples in the cluster. In some initial tests, we have observed an up to around 20% reduction in lighting computations, at very little computational cost. (However, this produced some subtle visual discrepancies, which we have been unable to work around at this point.)

We believe that it is possible to produce high quality approximations. These approximations may require additional per-cluster data, such as average shininess for specular computations. A better heuristic for determining when approximation is possible would also have to be developed.

It would also be interesting to investigate how clustered shading interacts with more complex shading, e.g. switching due to type of material. Since clustered shading has a much smaller shading cost than tiled shading, we expect better scaling with shader complexity.

References

[AA03] Arvo J., Aila T.: Optimized shadow mapping using the stencil buffer. *journal of graphics, gpu, and game tools 8*, 3 (2003), 23–32. 6

[And09] Andersson J.: Parallel graphics in frostbite - current & future. SIGGRAPH Course: Beyond Programmable Shading, 2009. URL: http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf. 2

[BE08] Balestra C., Engstad P.-K.: The technology of uncharted: Drake's fortune. Game Developer Conference, 2008. URL: http://www.naughtydog.com/docs/Naughty-Dog-GDC08-UNCHARTED-Tech.pdf. 2

[BOA09] Billeter M., Olsson O., Assarsson U.: Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 159–166. doi:http://doi.acm.org/10.1145/1572769.1572795. 4

[Cof11] Coffin C.: Spu-based deferred shading in battlefield 3 for playstation 3. GDC 2011, 2011. URL: http://www.slideshare.net/DICEStudio/spubased-deferred-shading-in-battlefield.-3-for-playstation-3. 2

[Cry10] Cryengine3 | crytek | sponza model, 2010. URL: http://www.crytek.com/cryengine/cryengine3/downloads. 7

[DWS*88] Deering M., Winner S., Schediwy B., Duffy C., Hunt N.: The triangle processor and normal vector shader: a vlsi system for high performance graphics. *SIGGRAPH Comput. Graph. 22*, 4 (1988), 21–30. doi:http://doi.acm.org/10.1145/378456.378468. 2

[Epi11] Epic Games: Unreal development kit, 2011. URL: http://www.udk.com/. 7

[ESSL10] Enderton E., Sintorn E., Shirley P., Luebke D.: Stochastic transparency. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), ACM, pp. 157–164. doi:http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1730804.1730830. 2

[FC11] Ferrier A., Coffin C.: Deferred shading techniques using frostbite in "battlefield 3" and "need for speed the run". In *ACM SIGGRAPH 2011 Talks* (New York, NY, USA, 2011), SIGGRAPH '11, ACM, pp. 33:1–33:1. doi:10.1145/2037826.2037869. 2

[GL10] Garanzha K., Loop C.: Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298. doi:10.1111/j.1467-8659.2009.01598.x. 2, 3

[HB10] Hoberock J., Bell N.: Thrust: A parallel template library, 2010. Version 1.3.0. URL: http://www.meganewtons.com/. 4, 7

[HM08] Hunt W., Mark W. R.: Ray-specialized acceleration structures for ray tracing. In *IEEE/EG Symposium on Interactive Ray Tracing 2008* (Aug 2008), IEEE/EG, pp. 3–10. 3

© The Eurographics Association 2012.

[HPLdW10] HOLLEMEERSCH C.-F., PIETERS B., LAMBERT P., DE WALLE R. V.: Accelerating virtual texturing using cuda. In *GPU Pro*, Engel W., (Ed.). A K Peters, 2010, pp. 623–642. 2

[Lau10] LAURITZEN A.: Deferred rendering for current and future rendering pipelines. SIGGRAPH Course: Beyond Programmable Shading, 2010. URL: http://bps10.idav.ucdavis.edu/talks/12-lauritzen_DeferredShading_BPS_SIGGRAPH2010.pdf. 2

[LD12] LIKTOR G., DACHSBACHER C.: Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 143–150. doi:10.1145/2159616.2159640. 3

[LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph. 25*, 1 (Jan. 2006), 60–99. URL: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1122501.1122505, doi:10.1145/1122501.1122505. 4

[LSO07] LEFOHN A. E., SENGUPTA S., OWENS J. D.: Resolution-matched shadow maps. *ACM Trans. Graph. 26*, 4 (2007), 20. doi:http://doi.acm.org/10.1145/1289603.1289611. 2

[LTYM06] LLOYD D. B., TUFT D., YOON S.-E., MANOCHA D.: Warping and partitioning for low error shadow maps. In *Proceedings of the Eurographics Workshop/Symposium on Rendering, EGSR* (June 2006), Eurographics Association, pp. 215–226. 4

[May10] MAYER A. J.: *Virtual Texturing*. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Oct. 2010. URL: http://www.cg.tuwien.ac.at/research/publications/2010/Mayer-2010-VT/. 2

[McK12] MCKEE J.: Technology behind amd's "leo demo". Game Developers Conference, 2012. URL: http://developer.amd.com/gpu_assets/AMD_Demos_LeoDemoGDC2012.ppsx. 2

[OA11] OLSSON O., ASSARSSON U.: Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4 (2011), 235–251. doi:10.1080/2151237X.2011.621761. 1, 2, 6, 7

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 197–206. doi:http://doi.acm.org/10.1145/97880.97901. 2

[Swo09] SWOBODA M.: Deferred lighting and post processing on playstation 3. Game Developer Conference, 2009. URL: http://www.technology.scee.net/files/presentations/gdc2009/DeferredLightingandPostProcessingonPS3.ppt. 2

© The Eurographics Association 2012.

Paper III: Efficient Virtual Shadow Maps for Many Lights

Ola Olsson, Erik Sintorn, Viktor Kämpe, Markus Billeter and Ulf Assarsson

Abstract: Recently, several algorithms have been introduced that enable real-time performance for many lights in applications such as games. In this paper, we explore the use of hardware-supported virtual cube-map shadows to efficiently implement high-quality shadows from hundreds of light sources in real time and within a bounded memory footprint. In addition, we explore the utility of ray tracing for shadows from many lights and present a hybrid algorithm combining ray tracing with cube maps to exploit their respective strengths.

I3D '14: Proceedings of the 2014 symposium on Interactive 3D graphics and games, to appear, March, 2014

Efficient Virtual Shadow Maps for Many Lights

Ola Olsson* Erik Sintorn* Viktor Kämpe* Markus Billeter* Ulf Assarsson*

Chalmers University of Technology

Figure 1: *Scenes rendered with many lights casting shadows at 1920×1080 resolution on an NVIDIA Geforce Titan. From the left:* HOUSES *with 1.01M triangles and 256 lights (23ms),* NECROPOLIS *with 2.58M triangles and 356 lights (34ms),* CRYSPONZA *with 302K triangles and 65 lights (16ms).*

Abstract

Recently, several algorithms have been introduced that enable real-time performance for many lights in applications such as games. In this paper, we explore the use of hardware-supported virtual cube-map shadows to efficiently implement high-quality shadows from hundreds of light sources in real time and within a bounded memory footprint. In addition, we explore the utility of ray tracing for shadows from many lights and present a hybrid algorithm combining ray tracing with cube maps to exploit their respective strengths. Our solution supports real-time performance with hundreds of lights in fully dynamic high-detail scenes.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms

Keywords: real-time,shadows,virtual,cube map

1 Introduction

In recent years, several techniques have been presented and refined that enable real-time performance for applications such as games using hundreds to many thousands of lights. These techniques work by binning lights into tiles of various dimensionality [Olsson and Assarsson 2011; Harada 2012; Olsson et al. 2012]. Many simultaneous lights enable both a higher degree of visual quality and greater artistic freedom, and these techniques are therefore directly applicable in the games industry [Swoboda 2009; Ferrier and Coffin 2011; Persson and Olsson 2013].

However, this body of previous work on real-time many-light algorithms has studied almost exclusively lights that do not cast shadows. While such lights enable impressive dynamic effects and more detailed lighting environments, they are not sufficient to capture the details in geometry, but tend to yield a flat look. Moreover, neglecting shadowing makes them more difficult to use, as light may leak through walls and similar occluding geometry, if care is not taken when placing the lights. For dynamic effects in interactive environments, controlling this behaviour is even more problematic. Shadowing is also highly important if we wish to employ the lights to visualize the result of some light-transport simulation, for example as done in *Instant Radiosity* [Keller 1997].

This paper aims to compute shadows for use in real-time applications supporting several tens to hundreds of shadow-casting lights. The shadows are of high and uniform quality, while staying within a bounded memory footprint.

As a starting point, we use *Clustered Deferred Shading* [Olsson et al. 2012], as this algorithm offers the highest light-culling efficiency among current real-time many-light algorithms and the most robust shading performance. This provides a good starting point when adding shadows, as the number of lights that require shadow computations is already close to the minimal set. Moreover, clustered shading provides true 3D bounds around the samples in the frame buffer and therefore can be viewed as a fast voxelization of the visible geometry. Thus, as we will see, clusters provide opportunities for efficient culling of shadow casters and allocation of shadow resolution.

1.1 Contributions

We contribute an efficient culling scheme, based on clusters, which is used to render shadow-casting geometry to many cube shadow maps. We demonstrate that this can enable real-time rendering performance using shadow maps for hundreds of lights, in dynamic scenes of high complexity.

We also contribute a method for quickly determining the required resolution of the shadow maps. This is used to show how hardware-supported virtual shadow maps may be efficiently implemented. To this end, we also introduce a very efficient way to determine the parts of the virtual shadow map that need physical backing. We demonstrate that these methods enable the memory requirements to stay within a limited range, while enabling uniform shadow quality.

Additionally, we explore the performance of ray tracing for many lights. We demonstrate that a hybrid approach, combining ray tracing and cube maps, offers high efficiency, in many cases better than

*e-mail:ola.olsson|erik.sintorn|kampe|billeter|uffe@chalmers.se

using either shadow maps or ray tracing individually.

We also contribute implementation details of the discussed methods, showing that shadow maps indeed can be made to scale to many lights. Thus, this paper provides an important benchmark for other research into real-time shadow algorithms for many lights.

2 Previous Work

Real Time Many Light Shading *Tiled Shading* is a recent technique that supports many thousands of lights in real-time applications [Swoboda 2009; Olsson and Assarsson 2011; Harada 2012]. In this technique, lights are binned into 2D screen-space tiles that can then be queried for shading. This is a very efficient and simple process, but the 2D nature of the algorithm creates a strong view dependence, resulting in poor worst case performance and unpredictable frame times.

Clustered Shading extends the technique by considering 3D bins instead, which improves efficiency and robustness [Olsson et al. 2012]. The clusters provide a three-dimensional subdivision of the view frustum and, thus, sample groupings with predictable bounds. This provides a basic building block for many of the new techniques described in this paper. See Section 3.1, for a more detailed overview.

Shadow Algorithms Studies on shadowing techniques generally present results using a single light source, usually with long or infinite range. Consequently, it is unclear how these techniques scale to many light sources, whereof a large proportion cover only a few samples. For a general review of shadow algorithms, see Eisemann et. al. [2011].

Virtual Shadow Maps Software-based virtual shadow maps have been explored in several publications to achieve high quality shadows in bounded memory [Fernando et al. 2001; Lefohn et al. 2007]. Recently, API and hardware extensions have been introduced that makes it possible to support virtual textures much more conveniently and with performance equalling that of traditional textures [Sellers et al. 2013].

Many light shadows There does exist a corpus of work in the field of real-time global illumination, which explores using many light sources with shadow casting, for example *Imperfect Shadow Maps* [Ritschel et al. 2008], and *Many-LODs* [Hollander et al. 2011]. However, these techniques generally assume that a large number of lights affect each sample to conceal approximation artifacts. In other words, these approaches are unable to produce accurate shadows for samples lit by only a few lights.

Ray Traced Shadows Recently, Harada et. al. [2013] described ray traced lights in conjunction with Tiled Forward Shading. They demonstrate that it can be feasible to ray trace shadows for many lights but do not report any analysis or comparison to other techniques.

3 Basic Algorithm

Our basic algorithm is shown below. The algorithm is constructed from clustered deferred shading, with shadow maps added. Steps that are inherited from ordinary clustered deferred shading are shown in gray.

1. Render scene to G-Buffers.
2. Cluster assignment – calculating the cluster keys of each view sample.
3. Find unique clusters – finding the compact list of unique cluster keys.
4. Assign lights to clusters. – creating a list of influencing lights for each cluster.
5. Select shadow map resolution for each light.
6. Allocate shadow maps.
7. Cull shadow casting geometry for each light.
8. Rasterize shadow maps.
9. Shade samples.

3.1 Clustered Shading Overview

In clustered shading the view volume is subdivided into a grid of self-similar sub-volumes (clusters), by starting from a regular 2D grid in screen space, e.g. using tiles of 32×32 pixels, and splitting exponentially along the depth direction. Next, all visible geometry samples are used to determine which of the clusters contain visible geometry. Once the set of occupied clusters has been found, the algorithm assigns lights to these, by intersecting the light volumes with the bounding box of each cluster. This yields a list of cluster/light pairs, associating each cluster with all lights that may affect a sample within (see Figure 2). Finally, each visible sample is shaded by looking up the lights for the cluster it is within and summing their contributions.

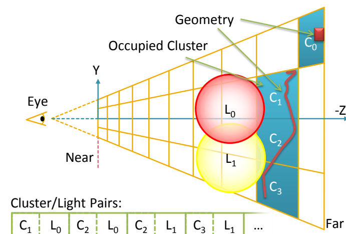


Figure 2: *Illustration of the depth subdivisions into clusters and light assignment. Clusters containing some geometry are shown in blue.*

The key pieces of information this process yields are a set of occupied clusters with associated bounding volumes (that approximate the visible geometry), and the near-minimal set of lights for each cluster. Intuitively, this information should be possible to exploit for efficient shadow computations, and this is exactly what we aim to do in the following sections.

3.2 Shadow Map Resolution Selection

One way to calculate the required resolution for each shadow map is to use the screen-space coverage of the light-bounding sphere. However, this produces vast overestimates whenever the camera is near, or within, the light volume. To calculate a more precisely matching resolution, one might follow the approach in *Resolution Matched Shadow Maps* (RMSM) [Lefohn et al. 2007], using shadow-map space derivatives for each view sample. However, applying this naïvely would be expensive, as the calculations must be repeated for each sample/light pair, and would require derivatives to be stored

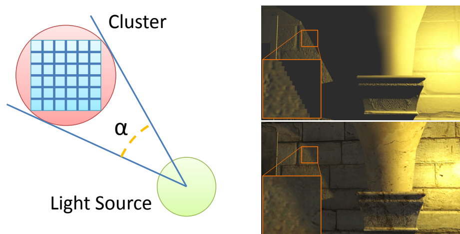


Figure 3: *Left, the solid angle of cluster, with respect to the light source, α, subtended by the cluster, illustrated in 2D. Right, example of undersampling due to an oblique surface violating assumptions in Equation 1, shown with and without textures and PCF.*

in the G-Buffer. Our goal is not to attempt alias-free shadows, but to quickly estimate a reasonable match. Therefore, we base our calculations on the bounding boxes of the clusters, which are typically several orders of magnitude fewer than the samples.

$$R = \sqrt{\frac{S/(\alpha/4\pi)}{6}} \tag{1}$$

The required resolution (R) for each cluster is estimated as the number of pixels covered by the cluster in screen space (S), divided by the proportion of the unit sphere subtended by the solid angle of the cluster bounding sphere (α), and distributed over the six cube faces (see Figure 3 and Equation 1).

This calculation is making several simplifying assumptions. The most significant is that we assume that the distribution of the samples is the same in shadow-map space as in screen space. This leads to an underestimate of the required resolution when the light is at an oblique angle to the surface (see Figure 3). A more detailed calculation might reduce these errors, but we opted to use this simple metric, which works well for the majority of cases.

For each cluster/light pair, we evaluate Equation 1 and retain the maximum R for each light as the shadow map resolution, i.e. a cube map with faces of resolution $R \times R$.

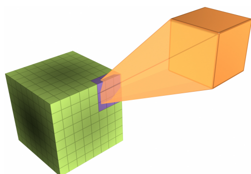


Figure 4: *The projected footprint (purple) of an AABB of either a batch or a cluster (orange), projected onto the cube map (green). The tiles on the cube map represent either virtual texture pages or projection map bits, depending on application.*

3.3 Shadow Map Allocation

Using the resolutions computed in the previous step, we can allocate one *virtual* cube shadow map for each light requiring a non-zero resolution. This does not allocate any actual physical memory backing the texture, just the virtual range.

In virtual textures, the pages are laid out as tiles of a certain size (e.g. 256×128 texels), covering the texture. Before we can render into the shadow map we must *commit* physical memory for those pages that will be sampled during shading. This can be established by projecting each sample onto the cube map, and record the requested page. To implement this efficiently, we again use the cluster bounds as proxy for the view samples, and project these onto the cube maps, (see Figure 4). The affected tiles are recorded in the *virtual-page mask*.

3.4 Culling Shadow-Casting Geometry

When managing many lights, culling efficiency is an important problem. The basic operation we wish to perform is to gather the minimal set of triangles that need to be rendered into each cube shadow map. This can be achieved by querying an acceleration structure with the bounding sphere defined by the light position and range. Real-time applications typically support this kind of query against a scene graph, or similar, for view frustum and shadow-map culling.

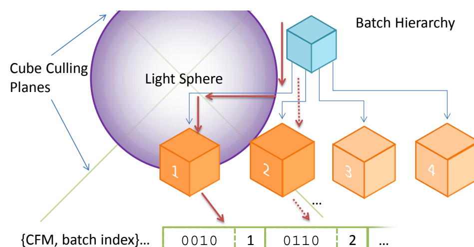


Figure 5: *Illustration of batch hierarchy traversal. The AABBs of batches* 1 *and* 2 *intersect the light sphere, and are tested against the culling planes, which determine the cube faces the batch must be rendered to.*

We make use of a *bounding volume hierarchy* (BVH), storing groups of triangles called *Batches* at the leaves. Each batch is represented by an *axis aligned bounding box* (AABB), which is updated at run time, and has a fixed maximum size. This allows us to explore which granularity offers the best performance for our use case. The hierarchy is queried for each light, producing a list of batch and light index pairs, identifying the batches to be drawn into each shadow map. For each pair, we record the result of culling for each cube face, as this information is needed later when rendering. The result is a bit mask with six bits that we call the *cube-face mask* (CFM), see Figure 5.

4 Algorithm Extensions

4.1 Projection Maps

Efficient culling also ought to avoid drawing geometry into unsampled regions of the shadow map. In other words, we require something that identifies where shadow receivers are located. This is similar in spirit to *projection maps*, which are used to guide photon distribution in photon maps, and we adopt this name.

Fortunately, this is almost exactly the same problem as establishing the needed pages for virtual textures (Section 3.3), and we reuse the method of projecting AABBs onto the cube faces. To represent the shadow receivers, each cube face stores a 32×32 bit mask (in contrast to page masks, which vary with resolution), and we rasterize the cluster bounds into this mask as before.

We then perform the same projection for each batch AABB that was found during the culling, to produce a mask for each shadow caster. If the logical intersection between these two masks is zero for any cube face, we do not need to draw the batch into this cube face. In addition to testing the mask, we also compute the maximum depth for each cube face and compare these to the minimum depth of each batch. This enables discarding shadow casters that lie behind any visible shadow receiver. For each batch, we update the cube-face mask to prune non-shadowing batches.

4.2 Non-uniform Light Sizes

The resolution selection presented in Section 3.2 uses the maximum sample density required by a cluster affected by a light. If the light is large and the view contains samples requiring very different densities, this can be a large over-estimate. This happens when a large light affects both some, relatively few, samples nearby the viewer but also a large portion of the visible scene further away (see Figure 6). The nearby samples dictate the resolution of the shadow map, which then must be used by all samples.

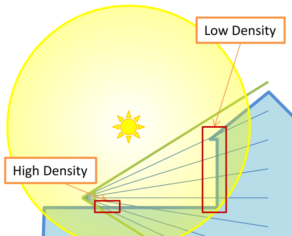


Figure 6: *Illustration of light requiring different sample densities within the view frustum. The nearby, high density, clusters dictate the resolution for the entire light.*

If there are only uniformly sized lights and we are comfortable with clamping the maximum allowed resolution, then this is not a significant problem. However, as our results show, if we have a scene with both large and small lights, then this can come to dominate the memory allocation requirements (e.g.NECROPOLIS, see Figure 12).

To eliminate this issue, we allow each light to allocate a number of shadow maps. We use a fixed number, as this allows fast and simple implementation, in our tests ranging from 1 to 16 shadow maps per light. To allocate the shadow maps, we add a step where we build a histogram over the resolutions requested by the clusters affecting each light. The maximum value within each histogram bucket is then used to allocate a distinct shadow map. When the shadow-map index is established, we replace the light index in the cluster light list with this index. Then, culling and drawing can remain the same, except that we sometimes must take care to separate the light index from the shadow-map index.

4.3 Level of Detail

For high-resolution shadow maps that are used for many view samples, we expect that rasterizing triangles is efficient, producing many samples for each triangle. However, low-resolution shadow maps sample the shadow-casting geometry sparsely, generating few samples per triangle. To maintain efficiency in these cases, some form of *Level of Detail* (LOD) is required.

In the limit, a light might only affect a single visible sample. Thus, it is clear that no amount of polygon-based LOD will suffice by itself. Consequently, we explore the use of ray tracing, which can random access geometry efficiently. To decide when ray tracing should be used, we simply use a threshold (in our tests we used 96 texels as the limit) on the resolution of the shadow map, which is tested after the resolution has been calculated. Those shadow maps that are below the threshold are not further processed and are replaced by directly ray tracing the shadows in a separate shading pass. We refer to this as the hybrid algorithm. Additionally, we evaluate using ray tracing for all shadows to determine the cross-over point in efficiency versus shadow maps.

Since we aim to use the ray tracing for LOD purposes, we chose to use a voxel representation, which has an inherent polygon-agnostic LOD and enables a much smaller memory footprint than would be possible using triangles. We use the technique described by Kämpe et. al. [2013], which offers performance comparable to state of the art polygon ray tracers and a very compact representation.

One difficulty with ray tracing is that building efficient acceleration structures is still a relatively slow process, at best offering interactive performance, and dynamically updating the structure is both costly and complex to implement [Karras and Aila 2013]. We therefore use a static acceleration structure, enabling correct occlusion from the static scene geometry, which often has the highest visual importance. As we aim to use the ray tracing for lights far away (and therefore low resolution), we consider this a practical use case to evaluate. For highly dynamic scenes, our results that use ray tracing are not directly applicable. Nevertheless, by using a high-performance accelerations structure, we aim to explore the upper bound for potential ray tracing performance.

To explore the use of polygon-based LOD, we evaluate a low-polygon version of the HOUSES scene (see Section 6). This is done in lieu of a full blown LOD system to attempt to establish an upper bound for shadow-mapping performance when LOD is used.

4.4 Explicit Cluster Bounds

As clusters are defined by a location in a regular grid within the view frustum, there is an associated bounding volume that is implied by this location. Computing explicit bounds, i.e. tightly fitting the samples within the cluster, was found by Olsson et al. [2012] to improve light-culling efficiency, but it also incurred too much overhead to be worthwhile. When introducing shadows and virtual shadow map allocation, there is more to gain from tighter bounds. We therefore present a novel design that computes approximate explicit bounds with very little overhead on modern GPUs.

We store one additional 32-bit integer for each cluster, which is logically divided into three 10-bit fields. Each of these represent the range of possible positions within the implicit AABB. With this scheme, the explicit bounding box can be established with just a single 32-bit `atomicOr` reduction for each sample. By using the bits to represent a number line, we can only represent as many discrete positions as there are bits. Thus, 10 bits for each axis enables down to a 1000-fold reduction in volume.

To reconstruct the bounding box, we make use of intrinsic bit-wise functions to count zeros from both directions in each 10-bit field. These bit positions are then used to scale and bias the implicit AABB in each axis direction.

5 Implementation

We implemented the algorithm and variants above using OpenGL and CUDA. All computationally intense stages are implemented on the GPU, and in general, we attempt to minimize stalls and GPU to CPU memory transfers. However, draw calls and rendering state

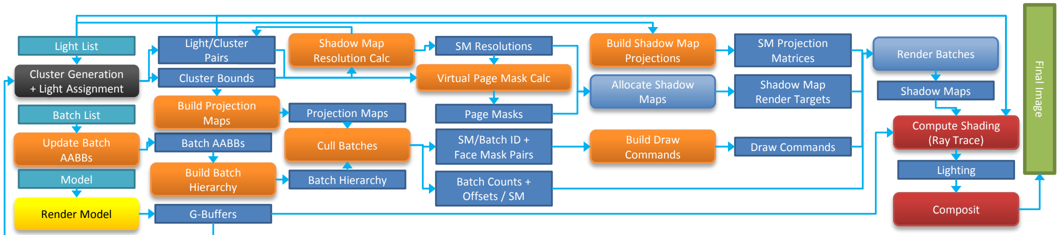


Figure 7: *Stages (rounded) and data (square) in the algorithm implementation. Stage colors correspond to those used in Figure 8. All computationally demanding stages are executed on the GPU, with sequencing and command issue performed by the CPU.*

changes are still necessary to invoke from the CPU, and thus, we must transfer some key information from the GPU. The system is illustrated in Figure 7.

5.1 Shadow Map Resolution Selection

The implementation of shadow-map resolution selection is a set of CUDA kernels, launched with one thread per cluster/light pair. These kernels compute the resolution, cube-face mask, virtual-page mask, and also the projection map, for each shadow map. To reduce the final histograms and bit masks, we use atomic operations, which provide adequate performance for current GPUs. The resulting array of shadow-map resolutions and the array of virtual-page masks are transferred to the CPU using an asynchronous copy.

5.2 Culling Shadow-Casting Geometry

In the implementation, we perform culling before allocating shadow maps, as this allows better asynchronous overlap, reducing stalls, and also minimizes transitions between CUDA and OpenGL operation.

5.2.1 Batch Hierarchy Construction

Each batch is a range of triangle indices and an AABB. A batch is constructed such that all the vertices share the transformation matrix[1] and are located close together, to ensure coherency under animation. At run time, we re-calculate each batch AABB from the vertices every frame to support animation. The resulting list is sorted along the Morton curve, and we then build an implicit left balanced 32-way BVH by recursively grouping 32 consecutive AABBs into a parent node. This is the same type of hierarchy that was used for hierarchical light assignment in clustered shading, and has been shown to perform well for many light sources [Olsson et al. 2012].

The batches are created off-line, using a bottom-up agglomerative tree-construction algorithm over the scene triangles, similar to that described by Walter et. al. [2008]. Unlike them, who use the surface area as the *dissimilarity function*, we use the length of the diagonal of the new cluster, as this produces more localized clusters (by considering all three dimensions). After tree construction, we create the batches by gathering leaves in sub-trees below some predefined size, e.g. 128 triangles (we tested several sizes, as reported below). The batches are stored in a flat array and loaded at run time.

5.2.2 Hierarchy Traversal

To gather the batches for each shadow map, we launch a kernel with a CUDA *block* for each shadow map. The reason for using blocks is that a modern GPU is not fully utilized when launching just a warp per light (as would be natural with our 32-way trees). The block uses a cooperative depth-first stack to utilize all warps within the block. We run this kernel in two passes to first count the number of batches for each shadow map and allocate storage, and then to output the array of batch indices. In between, we also perform a prefix sum to calculate the offsets of the batches belonging to each shadow map in the result array. We also output the cube-face mask for each batch. This mask is the bitwise `and` between the cube-face mask of the shadow map and the batch. The counts and offsets are copied back to the CPU asynchronously at this stage, as they are needed to issue drawing commands.

To further prune the list of batches, we launch another kernel that calculates the projection-map overlap for each batch in the output array and updates the cube-face mask.

The final step in the culling process is to generate a list of draw commands for OpenGL to render. We use the OpenGL 4.3 *multi-draw indirect* feature (`glMultiDrawElementsIndirect`), which allows the construction of draw commands on the GPU. We map a buffer from OpenGL to CUDA and launch a kernel where each thread transforms a batch index and cube-face mask output by the culling into a drawing command. The vertex count and offset is provided by the batch definition, and the instance count is the number of set bits in the cube-face mask.

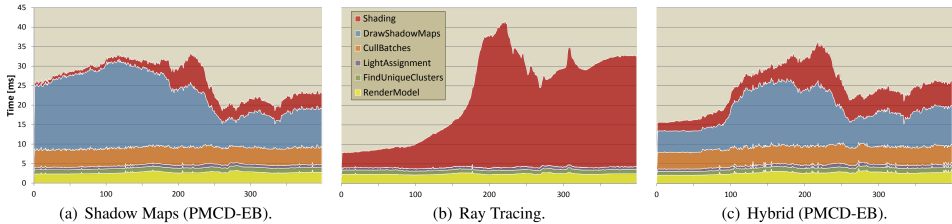
5.3 Shadow Map Allocation

To implement the virtual shadow maps, we make use of the OpenGL 4.4 ARB extension for sparse textures (`ARB_sparse_texture`). The extension enables vendor-specific page sizes which can be queried. Textures with sparse storage must be aligned to page boundaries. On our target hardware, the page size is 256×128 texels for 16-bit depth textures (64kb), which means that our square cube-map faces must be aligned to the larger value. For our implementation, the practical page granularity is therefore 256×256 texels, and this also limits the maximum resolution of our shadow maps to $8K \times 8K$ texels, as we use up to 32×32 bits in the virtual-page masks.

Thus, for each non-zero value in the array of shadow map resolutions, we round the requested resolution up to the next page boundary and then use this value to allocate a texture with virtual storage specified. Next, we iterate the virtual-page mask for each face and commit physical pages. If the requested resolution is small, in our implementation below 64×64 texels, we use an ordinary physical cube map instead.

In practice, allocating textures is a slow operation in OpenGL, and we instead pre-allocate a pool of cube textures. We create enough textures of each resolution to match the peak demands of our application. Since the textures are virtual (or small), the memory demands

[1] We only implement support for a single transform per vertex, but this is trivially extended to more general transformations, e.g. skinning.



(a) Shadow Maps (PMCD-EB). (b) Ray Tracing. (c) Hybrid (PMCD-EB).

Figure 8: *Timings from the* NECROPOLIS *scene animation. The performance is broken down into the principal stages of the algorithms. Note that for (b) and (c), the ray tracing time forms part of the shading.*

of this pool is small. At run time, we pick a cube map of the correct resolution from this pool and proceed as before.

5.3.1 Workarounds

Unfortunately, committing physical storage is very slow on current drivers[2]. As a fall back, we therefore implemented an additional pool of physical textures, and pick the next free one of the closest matching resolution. For the physical pool, we cannot allocate all the needed resolutions up-front, as the memory requirements are prohibitive, e.g. a single 8K cube map requires 750Mb of memory (this, in fact, being the raison d'être for the virtual shadow maps). Consequently, this method will suffer from very poor and varying shadow quality but enables us to measure the performance of all the other parts of the algorithm.

On game consoles, where the developers are able to directly manage resources, the straightforward implementation might be expected to work well. Also, extensions such as the explicit page-pool management proposed by AMD (`AMD_texture_tile_pool`) [Sellers et al. 2013] indicate that the page-allocation performance problem is possible to address. For our purposes, going yet further and allowing pages to be managed fully on the GPU, for example using some manner of *indirect* call, similar to that used for draw commands, would seem ideal.

5.4 Rasterizing Shadow Caster Geometry

With the set up work done previously, the actual drawing is straightforward. For each shadow map, we invoke `glMultiDrawElementsIndirect` once, using the count and offset shipped back to the CPU during the culling. To route the batches to the needed cube map faces, we use layered rendering and a geometry shader. The geometry shader uses the instance index and the cube-face mask (which we supply as a per-instance vertex attribute) to compute the correct layer.

The sparse textures, when used as a frame buffer target, quietly drop any fragments that end up in uncommitted areas. This matches our expectations well, as such areas will not be used for shadow look ups. Compared to previous work on software virtual shadow maps, this is an enormous advantage, as we sidestep the issues of fine-grained binning, clipping and copying and also do not have to allocate temporary rendering buffers.

We did not implement support for different materials (e.g. to support alpha masking). To do so, one draw call per shadow material type would be needed instead.

[2]The NVIDIA beta driver version 327.24 was used in our measurements.

6 Results and Discussion

All experiments were conducted on an NVIDIA GTX Titan GPU and an Intel Core i7-3930K CPU. We used three scenes (see Figure 1). HOUSES is designed to be used to illustrate the scaling in a scene where all lights have a similar size and uniform distribution. NECROPOLIS is derived from the Unreal SDK, with some lights moved slightly and all ranges doubled. We added several animated cannons shooting lights across the main central area, and a number of moving objects. The scene contains 275 static lights and peaks at 376 lights. CRYSPONZA is derived from the Crytek version of the Sponza atrium scene, with 65 light sources added. Each scene has a camera animation, which is used in performance graphs (see the supplementary video).

We evaluate several algorithm variants with different modifications: Shadow maps with projection map culling (*PMC*), and with added depth culling (*PMCD*); with or without explicit bounds (*EB*); only using cluster face mask culling (*CFM*); Ray Tracing; and Hybrid, which uses PMCD-EB. Unless otherwise indicated, four cube shadow maps per light is used.

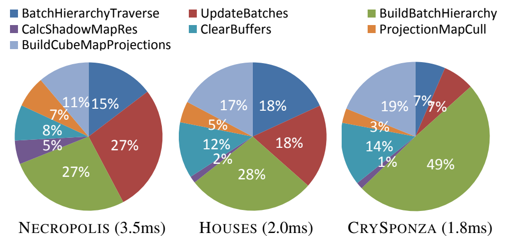
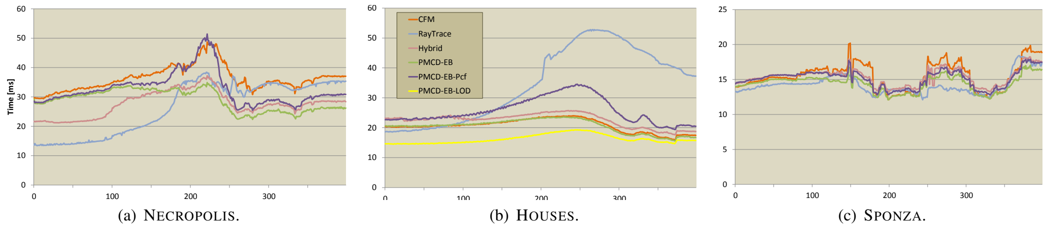


Figure 9: *Timing breakdown of the steps involved in culling batches. The displayed percentage represents the maximum time for each of the steps over the entire animation.*

As noted in Section 5.3.1, current API and driver performance for committing physical memory is very poor. All performance measurements are therefore reported using the fall-back implementation, which uses a pool of physical pre-allocated shadow maps. We performed the same measurements on the full implementation to ensure that they produce representative figures. The pool will run out of high-resolution shadow maps at times, which results in too low sample density and affects the shadow map rendering times. These variations are within 100% of the reported figures and do not affect the peak times reported. It was found that other factors such as re-binding render targets had greater performance impact.



(a) NECROPOLIS. (b) HOUSES. (c) SPONZA.

Figure 10: *Wall-to-wall frame times from the scene animations, for different algorithm variations.*

All reported figures are using a batch size of up to 128 triangles. We evaluated several other batch sizes and found that performance was similar in the range 32 to 512 triangles per batch, but was significantly worse for larger batches. This is expected, as larger batches lead to more triangles being drawn, and rasterization is already a larger cost than culling in the algorithm (see Figure 8(a)).

Performance We report the wall-to-wall frame times for our main algorithm variants in Figure 10. These are the times between consecutive frames and thus include all rendering activity needed to produce each frame. From these results, it is clear that virtual shadow maps with projection-map culling offer robust and scalable performance and that real-time performance with many lights and dynamic scenes is achievable.

As expected, ray tracing offers better scaling when the shadows require fewer samples, with consistently better performance in the first part of the zooming animations in NECROPOLIS and HOUSES (Figure 10). When the lights require more samples, shadow maps generally win, and also provide better quality (as we are ray tracing a fairly coarse voxel representation).

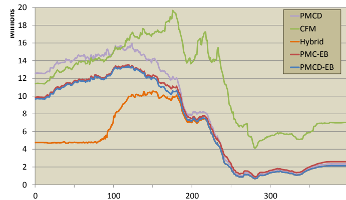


Figure 11: *Triangles drawn each frame in the* NECROPOLIS *animation with different culling methods. The naïve method, that is, not using the information about clusters to improve culling, is not included in the graph to improve presentation. It renders between 40 and 126 million triangles per frame and never less than six times the number of PMCD.*

The hybrid method is able to make use of this advantage and provides substantially better performance early in the NECROPOLIS animation (Figure 8(c)). However, it fails to improve worst-case performance because there are always a few small lights visible, and our implementation runs a separate full-screen pass in CUDA to shade these. Thus, efficiency in these cases is low, and we would likely see better results if the ray tracing better integrated with the other shading. An improved selection criterion, based on the estimated cost of the methods rather than just shadow-map resolution, could also improve performance. For example, the LOD version of the HOUSES scene (Figure 10(b)) highlights that the cost of shadow mapping is correlated to the number of polygons rendered. The LOD version also demonstrates that there exists a potential for performance improvements using traditional polygon LOD, as an alternative or in addition to ray tracing.

Shadow filtering, in our implementation a simple nine-tap *Percentage-Closer filter* (PCF), has a quite high proportion of the total cost, especially in the scenes with relatively many lights affecting each sample (Figure 10). Thus, techniques to reduce this cost, by restricting filtering or using pre-filtering, could be a useful addition.

Culling Efficiency Culling efficiency is greatly improved by our new methods exploiting information about shadow receivers inherent in the cluster, as shown in Figure 11. Compared to naïvely culling using the light sphere and drawing to all six cube faces, our method is at least six times more efficient.

When adding the max depth culling for each cube face, the additional improvement is not as significant. This is not unexpected as the single depth is a very coarse representation, most lights are relatively short range, and the scene is mostly open with little occlusion. Towards the end of the animation, where the camera is inside a building, the proportion that is culled by the depth increases somewhat. The cost of adding this test is very small (see Figure 9: 'ProjectionMapCull').

Memory Usage As expected, using only a single shadow map per light has very high worst case for NECROPOLIS (Figure 12:'PMCD-EB-1SM'). With four shadow maps per light, we get a better correspondence between lighting computations (i.e., the number of light/sample pairs shaded) and number of shadow maps texels allocated. This indicates that peak shadow map usage is correlated to the density of lights in the scene, which is a very useful property when budgeting rendering resources. The largest number of shadow-map texels per lighting computation occurs when shadow maps are low resolution, early in the animation, and does not coincide with peak memory usage. We tested up to 16 shadow maps per light, and above eight, the number of texels rises again.

Explicit bounds The explicit bounds provide improved efficiency for both the number of shadow-map texels allocated and number of triangles drawn by $8 - 35\%$ over the NECROPOLIS animation. The greatest improvement is seen near the start of the animation, where many clusters are far away and thus have large implicit bounds in view space (Figure 11).

Quality As seen in Figure 3, there exist sampling artifacts due to our choice of resolution calculations. However, as we recalculate the required resolutions continuously and select the maximum for each shadow map, we expect these errors to be stable and consistent. In

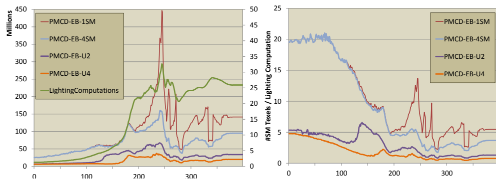


Figure 12: *Allocated shadow-map texels for various scenarios over the* NECROPOLIS *animation. Shows the performance with a varying number of shadow maps per light, the effect of the global undersampling parameter (u2|u4 suffix), and also plots the number of Lighting Computations for each frame (secondary axis).*

the supplementary video, it is difficult to notice any artifacts caused by switching between shadow-map resolutions.

We also added a global parameter controlling undersampling to enable trading visual quality for lower memory usage (see Figure 12). This enables a lower peak memory demand with uniform reduction in quality. For a visual comparison, see the supplementary video.

7 Conclusion

We presented several new ways of exploiting the information inherent in the clusters provided by clustered shading, which enable very efficient and effective culling of shadow casting geometry. With these extensions, we have demonstrated that using hardware-supported virtual cube shadow maps is a viable method for achieving high-quality real-time shadows, scaling to hundreds of lights.

In addition, we show that memory requirements when using virtual cube shadow maps as described in this paper remains proportional to the number of shaded samples. This is again enabled by utilizing clusters to quickly determine both the resolution and coverage of the shadow maps.

We also demonstrate that using ray tracing can be more efficient than shadow maps for shadows with few samples and that a hybrid method building on the strength of both is a promising possibility.

The implementation of `ARB_sparse_texture` used in our evaluation does not offer real-time performance. However, we expect that future revisions, perhaps combined with new extensions, will make this possible. In addition, on platforms with more direct control over resources, such as game consoles, this problem should be greatly mitigated.

8 Future Work

In the future, we would like to explore more aggressive culling schemes, for example using better max-depth culling. We also would like to explore other light distributions, which might be supported by pre-defined masks, yielding high flexibility in distribution.

Acknowledgements

The Geforce GTX Titan used for this research was donated by the NVIDIA Corporation. We also want to acknowledge the anonymous reviewers for their valuable comments, and Jeff Bolz, Piers Daniell and Carsten Roche of NVIDIA for driver support. This research was supported by the Swedish Foundation for Strategic Research under grant RIT10-0033.

References

EISEMANN, E., SCHWARZ, M., ASSARSSON, U., AND WIMMER, M. 2011. *Real-Time Shadows*. A.K. Peters.

FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2001. Adaptive shadow maps. In *Proc.*, SIGGRAPH '01, 387–390.

FERRIER, A., AND COFFIN, C. 2011. Deferred shading techniques using frostbite in "battlefield 3" and "need for speed the run". In *Talks*, SIGGRAPH '11, 33:1–33:1.

HARADA, T., MCKEE, J., AND YANG, J. C. 2013. Forward+: A step toward film-style shading in real time. In *GPU Pro 4: Advanced Rendering Techniques*, W. Engel, Ed. 115–134.

HARADA, T. 2012. A 2.5D culling for forward+. In *SIGGRAPH Asia 2012 Technical Briefs*, SA '12, 18:1–18:4.

HOLLANDER, M., RITSCHEL, T., EISEMANN, E., AND BOUBEKEUR, T. 2011. ManyLoDs: parallel many-view level-of-detail selection for real-time global illumination. *Computer Graphics Forum 30*, 4, 1233–1240.

KARRAS, T., AND AILA, T. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proc.*, HPG '13, 89–99.

KELLER, A. 1997. Instant radiosity. In *Proc.*, SIGGRAPH '97, 49–56.

KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2013. High resolution sparse voxel dags. *ACM Trans. Graph. 32*, 4. SIGGRAPH 2013.

LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution-matched shadow maps. *ACM Trans. Graph. 26*, 4 (Oct.).

OLSSON, O., AND ASSARSSON, U. 2011. Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4, 235–251.

OLSSON, O., BILLETER, M., AND ASSARSSON, U. 2012. Clustered deferred and forward shading. In *Proc.*, EGGH-HPG'12, 87–96.

PERSSON, E., AND OLSSON, O. 2013. Practical clustered deferred and forward shading. In *Courses: Advances in Real-Time Rendering in Games*, SIGGRAPH '13, 23:1–23:88.

RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., AND KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. 27*, 5 (Dec.), 129:1–129:8.

SELLERS, G., OBERT, J., COZZI, P., RING, K., PERSSON, E., DE VAHL, J., AND VAN WAVEREN, J. M. P. 2013. Rendering massive virtual worlds. In *Courses*, SIGGRAPH '13, 23:1–23:88.

SWOBODA, M., 2009. Deferred lighting and post processing on playstation 3. Game Developer Conference.

WALTER, B., BALA, K., KULKARNI, M., AND PINGALI, K. 2008. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008*, 81–86.

Paper IV: An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects using per-triangle Shadow Volumes

Erik Sintorn, Ola Olsson and Ulf Assarsson

Abstract: This paper presents a novel method for generating pixel-accurate shadows from point light-sources in real-time. The new method is able to quickly cull pixels that are not in shadow and to trivially accept large chunks of pixels thanks mainly to using the whole triangle shadow volume as a primitive, instead of rendering the shadow quads independently as in the classic Shadow-Volume algorithm. Our CUDA implementation outperforms z-fail consistently and surpasses z-pass at high resolutions, although these latter two are hardware accelerated, while inheriting none of the robustness issues associated with these methods. Another, perhaps even more important property of our algorithm, is that it requires no pre-processing or identification of silhouette edges and so robustly and efficiently handles arbitrary triangle soups. In terms of view sample test and set operations performed, we show that our algorithm can be an order of magnitude more efficient than z-pass when rendering a game-scene at multi-sampled HD resolutions. We go on to show that the algorithm can be trivially modified to support textured, semi-transparent and colored semi-transparent shadow-casters and that it can be combined with either depth-peeling or stochastic transparency to also support transparent shadow receivers. Compared to recent alias-free shadow-map algorithms, our method has a very small memory footprint, does not suffer from load-balancing issues, and handles omni-directional lights without modification. It is easily incorporated into any deferred rendering pipeline and combines many of the strengths of shadow maps and shadow volumes.

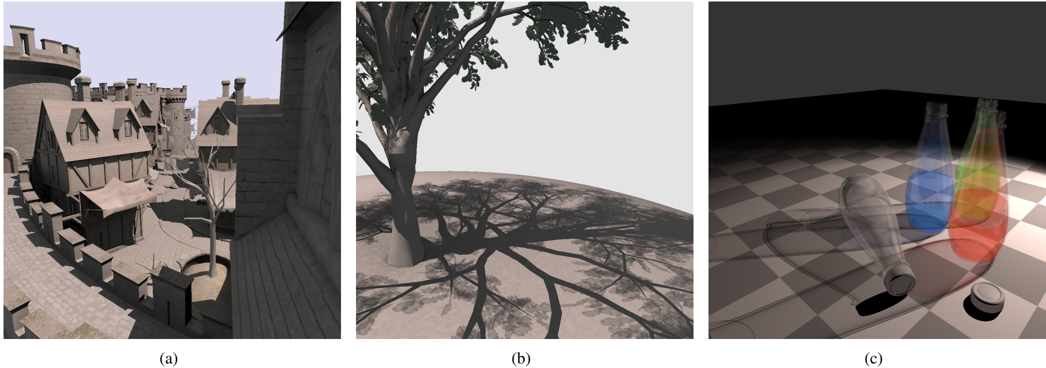
ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2011, Volume 30, Issue 6, Article No. 153, December 2011

An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects using per-triangle Shadow Volumes

Erik Sintorn*
Chalmers University Of Technology

Ola Olsson†
Chalmers University Of Technology

Ulf Assarsson‡
Chalmers University Of Technology



(a) (b) (c)

Figure 1: *Images rendered with the novel shadow algorithm. All images rendered in 1024x1024, time taken to generate shadow buffers in parenthesis. (a) Pixel accurate hard shadows in a game scene (7.29ms, 60k triangles). (b) Alpha-textured shadow casters (13ms, 35k triangles). (c) Colored transparent shadows. Image rendered using depth peeling of 8 layers (75.66ms, 5-19 ms per layer, 60k triangles).*

Abstract

This paper presents a novel method for generating pixel-accurate shadows from point light-sources in real-time. The new method is able to quickly cull pixels that are not in shadow and to trivially accept large chunks of pixels thanks mainly to using the whole triangle shadow volume as a primitive, instead of rendering the shadow quads independently as in the classic Shadow-Volume algorithm. Our CUDA implementation outperforms z-fail consistently and surpasses z-pass at high resolutions, although these latter two are hardware accelerated, while inheriting none of the robustness issues associated with these methods. Another, perhaps even more important property of our algorithm, is that it requires no pre-processing or identification of silhouette edges and so robustly and efficiently handles arbitrary triangle soups. In terms of view sample test and set operations performed, we show that our algorithm can be an order of magnitude more efficient than z-pass when rendering a game-scene at multi-sampled HD resolutions. We go on to show that the algorithm can be trivially modified to support textured, semi-transparent and colored semi-transparent shadow-casters and that it can be combined with either depth-peeling or stochastic transparency to also support transparent shadow receivers. Compared to recent alias-free shadow-map algorithms, our method has a very small memory footprint, does not suffer from load-balancing issues, and handles omni-directional lights without modification. It is easily incorporated into any deferred rendering pipeline and combines many of the strengths of shadow maps and shadow volumes.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: shadows, alias-free, real time, transparency

*e-mail: erik.sintorn@chalmers.se
†e-mail:ola.olsson@chalmers.se
‡e-mail:uffe@chalmers.se

Links: DL PDF

1 Introduction

Generating accurate shadows from point light-sources for each pixel sample remains a challenging problem for real-time applications. Despite generations of research, we have yet to see a pixel-accurate shadow-algorithm for point lights that requires no pre-processing, works on any arbitrary set of triangles and that runs at stable real-time frame rates for typical game-scenes on consumer level hardware. Traditional shadow mapping [Williams 1978] techniques generate shadows from a discretized image representation of the scene and so alias when queried for light visibility in screen space. Real-time techniques based on irregular rasterization [Sintorn et al. 2008] tend to generate unbalanced workloads that fit current GPUs poorly and consequently, frame rates are often very unstable. Real-time ray tracing algorithms rely heavily on geometry pre-processing to generate efficient acceleration structures. Finally, robust implementations of the Shadow-Volume algorithm require pre-processing the mesh to find edge connectivity, work poorly or not at all for polygon soups without connectivity, and have frame rates that are all but stable as the view of a complex scene changes. Nevertheless, the idea of directly rasterizing the volumes that represent shadows onto the view samples remains compelling. In the

ACM Reference Format
Sintorn, E., Olsson, O., Assarsson, U. 2011. An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects using per-triangle Shadow Volumes. *ACM Trans. Graph. 30*, 6, Article 153 (December 2011), 10 pages. DOI = 10.1145/2024156.2024187 http://doi.acm.org/10.1145/2024156.2024187.

Copyright Notice
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 0730-0301/2011/12-ART153 $10.00 DOI 10.1145/2024156.2024187
http://doi.acm.org/10.1145/2024156.2024187

upcoming sections, we hope to convince the reader that this basic idea is sound and that choosing the right manner of rasterization is the key to efficiently generate shadows using shadow volumes.

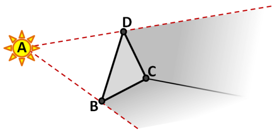


Figure 2: *A point lies in the shadow volume of a triangle if it lies behind all planes CAB, DAC, BAD and BCD. To optimize culling of tiles, let E be the eye position and test against the planes EDA and EAB, in order to get a less conservative tile-test (see Section 3.5) . These two planes are depicted with red dashed lines.*

The basic algorithm described in this paper can be easily summarized. We render the shadow volume of each triangle, see Figure 2, onto a depth buffer hierarchy generated from the standard depth buffer after rendering the camera view of the scene. A *node* in this hierarchy is a texel at some level, containing the min and max depths and defines a bounding box in normalized device coordinates (see Section 3). To avoid confusion with traditional shadow volumes, which represent shadows from an object, we will refer to our single triangle shadow volumes as *shadow frustums*. A shadow frustum can either intersect a node, in which case the child-nodes must be inspected; completely envelop a node, in which case the node and all its child-nodes are in shadow and can be flagged as such; or the node can lie completely outside the frustum, in which case all child-nodes can be abandoned. We have implemented this algorithm in CUDA, where it takes the form of a hierarchical rasterizer operating on shadow frustum primitives. The algorithm has many things in common with the traditional Shadow-Volume algorithm, but by considering every shadow frustum separately (as opposed to forming the shadow volume from the silhouette edges of an object) and by maintaining the complete frustum throughout traversal (as opposed to splitting it into per-edge *shadow-quads*), we manage to elegantly steer clear of the many quirks and robustness issues that are associated with other Shadow-Volume algorithms.

Our main contribution is an accurate and efficient algorithm for determining light source visibility for all view samples which:

- works for any arbitrary triangle-soup without pre-processing.
- is demonstrated to be very efficient in the amount of work performed per shadowed view sample.
- has a very low memory footprint.
- trivially extends to allow for textured shadow-casters.
- trivially extends to allow for semi-transparent and colored semi-transparent shadow-casters.
- is easy to integrate into any deferred rendering pipeline.

We show that the shadow frustum–depth hierarchy traversal, as well as interpolation of per-vertex attributes, can be done entirely in homogeneous clip space, thereby eliminating any potential problems with near and far clip planes. Additionally, we suggest a novel method where binary light visibility is recorded stochastically per view sample with a probability equal to the opacity of the shadow-casting triangle. This allows us to store a single bit of visibility information per sample in *Multi-Sample Anti Aliasing* (MSAA), *Coverage-Sample Anti Aliasing* (CSAA) or *Super-Sample Anti Aliasing* (SSAA) rendering contexts and still get a correct-on-average visibility result when the pixel is resolved. We also propose a scheme to anti-alias the shadow-edges without actually using more than one depth-value per pixel in the depth-hierarchy. This method is equivalent to PCF filtering of an infinite resolution shadow map and like PCF filtering gives occasionally incorrect but visually pleasing results.

2 Previous Work

For a thorough overview of real-time shadow algorithms, we refer the reader to [Eisemann et al. 2009]. Below, we will review the work most relevant to the algorithm presented in this paper. Algorithms for rendering hard shadows in real-time can be roughly sorted into three categories:

Shadow mapping Today shadow mapping [Williams 1978] and related techniques constitute the de-facto standard shadowing algorithm for real-time applications, despite suffering from quite severe aliasing artifacts. This widespread adoption has come about due to good hardware support, ability to handle arbitrary geometry, and low variability in frame times.

Another reason for the popularity is that the shadow map can be filtered to hide artifacts, mimicking the effect of an area light-source. Filtering during lookup usually requires a large number of samples [Reeves et al. 1987; Fernando 2005], whereas pre-filtering requires additional attributes, which increases storage and bandwidth requirements [Donnelly and Lauritzen 2006; Annen et al. 2008]. Filtering can enable the use of relatively low resolution shadow maps while producing visually pleasing results.

However, without resorting to a very high-resolution shadow map, sharp shadows cannot be produced, leaving shadows artificially blurred or with obvious discretization artifacts. Several algorithms attempt to improve precision where it is most needed, without abhorrent memory requirements, either by warping or by partitioning the shadow map. Warping techniques [Stamminger and Drettakis 2002; Wimmer et al. 2004; Lloyd et al. 2008] can yield impressive results but suffer from special cases where they degenerate to ordinary shadow maps. Partitioning approaches can produce high-quality sharp shadows quickly for some scenes [Arvo 2004; Zhang et al. 2006; Lefohn et al. 2007; Lauritzen et al. 2011], but have difficulties if the scenes are very open with widely distributed geometry, leading to aliasing re-appearing, unpredictable run-time performance, or escalating memory requirements.

Alias-free shadow maps Alias-free shadow-mapping algorithms are exact per view sample [Aila and Laine 2004; Johnson et al. 2005; Sintorn et al. 2008]. To our knowledge, the only pixel accurate alias-free shadow algorithm that runs in real-time on current GPUs for complex scenes is [Sintorn et al. 2008]. While this algorithm runs admirably on some scenes and is likely to perform as well as or better than ours on views with a very high variance in the depth buffer, it breaks down in other configurations (e.g. when all view samples project to a single line in light space). Like our algorithm, these exact methods could trivially support semi-transparent and textured shadow casters.

Shadow volumes The Shadow-Volume algorithm, introduced by Crow in [1977], was implemented with hardware acceleration in 1985 [Fuchs et al. 1985] but did not see widespread use until a version was suggested that could be hardware accelerated on consumer grade graphics hardware with the z-pass algorithm [Heidmann 1991]. The idea is to isolate the *silhouette-edges* and extrude these away from the light-source, forming *shadow-quads* that enclose the *shadow volume* for an object. These shadow volumes are

rendered onto the camera's depth buffer and the stencil buffer is incremented for front-facing quads and decremented for back-facing so that, when all quads are processed, the stencil buffer value will be 0 only for those pixels that do not lie in shadow. This essentially creates a per-pixel count of the number of shadow volumes that are entered by a ray cast from the eye to the view sample. Alternatively, the counting can be performed from the view samples to infinity, a method called *z-fail* [Bilodeau and Songy 1999; Carmack 2000].

Z-pass classically suffers from the eye-in-shadow problem, i.e. if the camera lies within one or more shadow volumes, or if the near-plane clips any shadow quad, the values in the stencil buffer will be incorrect. This is partly solved by Hornus et al. [2005] where the lights' view is aligned with that of the camera and the light's far plane is set to equal the camera's near plane. Since the light is not in shadow, the scene can then be rendered from the light with a projection matrix set up to match that of the camera, and the stencil buffer is updated with all shadow quads that lie between the light and the camera near plane in a first pass. The algorithm has some robustness issues that get worse as the lights' position approaches the cameras near plane. With the advent of *depth-clamping*, the solution to the eye in shadow problem is reduced to evaluating how many shadow casters lie between the camera and light, but to date no fully robust solution has been presented to this problem.

While the z-fail algorithm can be made practically robust [Everitt and Kilgard 2002], it is typically significantly slower due to higher overdraw [Laine 2005] and the need for near- and far capping geometry. An eight-bit stencil buffer (still the maximum allowed on current GPUs) can overflow, and resorting to using higher-precision color buffers will accentuate the rasterization cost significantly.

Several papers exist that aim to reduce fill rate requirements. In [Lloyd et al. 2004], the authors consider the objects in a scene graph and discuss a number of ways to prune the set of objects to find potential shadow-casters (for the current light view) and potential shadow receivers (for the current camera view). They also suggest a way to limit the distance a shadow caster has to extend its shadow volume in order to conservatively reach all potential shadow receivers. All of the optimizations in this paper are equally applicable to our algorithm (if we were to include a far plane for our shadow frustums), but as our rasterization already culls receiving tiles much more efficiently than the z-pass or z-fail algorithms, the overdraw reductions would probably not be large. Aila and Akenine-Moller [2004] suggest an optimization that in some ways resemble ours. In the first stage of their algorithm, 8x8 pixel tiles that lie on the shadow-volume boundary are identified (a minimum and maximum z for each tile is maintained, and thus, the 3D bounding box for the tile can be tested against each shadow-quad) and other tiles are classified as either fully in shadow or fully lit. One such shadow buffer (containing a boolean *boundary* flag and an eight-bit *stencil* value) is required per shadow volume. In the second stage, the per pixel shadow is calculated for boundary tiles whereas non-boundary tiles can be set to the shadow state of the tile. The authors suggest two hardware modifications to make the algorithm more efficient: a hierarchical stencil-buffer that would make classification of tiles faster and a so called Delay Stream that would help the rasterizer keep track of when the classification of a shadow volume is complete and final stencil buffer updates can begin. Nevertheless, this solution would be infeasible for a larger amount of shadow volumes.

Chan and Durand [2004] use a shadow map to fill in the hard shadows and also identify shadow-silhouette pixels. Then, shadow volumes are used to generate correct shadows at these silhouette pixels. However, the algorithm relies on custom hardware to reject non-silhouette pixels and cannot guarantee not missing small silhouettes due to the discrete shadow map sampling.

Textured and semi transparent shadows Materials that are semi-transparent are common in real-time applications and present a problem for most shadow algorithms. A semi transparent surface is given an *alpha* or *opacity* value, α, which is defined as the ratio of received light that is absorbed or reflected at the surface ($1 - \alpha$ is the ratio that continues through the surface). This same model is commonly used to represent both materials that have partial coverage (e.g. a screen door) and materials that transmit light (e.g. thin glass) [McGuire and Enderton 2011]. For shadow-map based methods, these materials are difficult because a visibility lookup is no longer a binary function and so, a single depth value is not sufficient to define the visibility along a ray from the light towards the point being shadowed. Several techniques have been suggested to solve this by rendering shadow maps with several layers, the most common being Deep Shadow Maps [Lokovic and Veach 2000] for which the scene has to be rendered several times in the absence of a hardware accelerated A-buffer [Carpenter 1984], or by sampling visibility at discrete depths [Kim and Neumann 2001; Yuksel and Keyser 2008; Sintorn and Assarsson 2009] which works well for "fuzzy" geometry such as smoke or hair, but not so well for polygons where strong transitions in visibility happen at every surface. Recently, a different method was proposed [Enderton et al. 2010; McGuire and Enderton 2011] in which, when generating the shadow map, a triangle fragment is simply discarded with probability $(1 - \alpha)$. A PCF lookup into this map will return a value that is correct on average. However, if too few taps are taken from the filter region, the result will be noisy and so, to achieve sharp and noise free shadows, very many taps will be required from a very high resolution shadow map.

Semi-transparent shadows have been considered for shadow-volume type algorithms as well. A straightforward approach was suggested in [Kim et al. 2008], where the stencil buffer is replaced by a floating point buffer and the stencil increment/decrement operations are replaced by adding or removing $log(1 - \alpha)$, where α is the opacity of the object that generated the shadow quad. This elegantly produces a final stencil value, s, such that $\exp(s) = \prod_i (1 - \alpha_i)$, where α_i is the opacity value of a shadow caster that covers the sample point, which is exactly the visibility at that point. The method produces pixel-accurate sharp shadows and can easily be extended to support colored shadow-casters, but requires opacity to be constant per object. Needless to say, the additional math and blending operations exacerbate the overdraw problem inherent in the traditional Shadow-Volume algorithm. Also, textured shadow casters are not supported.

In [Hasselgren and Akenine-Moller 2007] the problem is instead solved simply by handling semi-transparent or textured objects separately, so that every triangle shadow volume is rendered for such objects. Despite the optimizations discussed in that paper, this will cause very much overdraw and will be prohibitively slow for complex geometry. The realization from that paper, that transparent and textured shadow casters are easily supported when the triangles' shadow volumes are handled separately, is, however, the basis for these extensions to our algorithm which renders per-triangle shadow volumes very efficiently. It should also be mentioned that a similar idea was used for textured soft shadows [Forest et al. 2009].

In games and other real-time graphics applications, complex geometry is often approximated by mapping a binary *alpha mask* to simple geometry. A common example is the leaves on the tree in Figure 1(b). Casting shadows from such objects is simple when using shadow-mapping type algorithms, where, when rendering the shadow map, a fragment can simply be discarded if the alpha value is below some threshold. Unfortunately, as the filtered opacity value will not be binary even if the original alpha mask is, this technique introduces even more aliasing to the shadow-map algorithm. The problem with shadow casters of this kind, when using shadow-

volume type algorithms, is that when a shadow-quad is rendered over a view sample, we know only that the sample lies within the shadow volume of some *object* but we have no means of determining which *triangle* covers the sample, and so, we cannot do a lookup into the alpha mask to see if the point is truly in shadow. If we render triangle shadow volumes individually however (as in [Hasselgren and Akenine-Moller 2007]), we may pass along the uv coordinates of each vertex, as well as the vertices themselves, and then do a ray-triangle intersection test prior to updating the stencil buffer, or by some other means find the uv coordinates on the triangle.

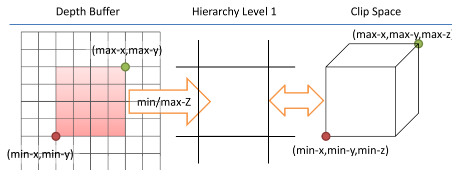


Figure 3: *Every texel in the second level of the depth hierarchy defines a bounding box in normalized device coordinates and, equivalently, a world space frustum which contains all view samples inside.*

3 Algorithm

We start out with as basic an approach to shadow volumes as we can imagine. The scene has been rendered to an off-screen set of buffers containing the ambient light component, direct lighting and the depth buffer. The depth buffer, along with the model-view-projection matrix, implicitly gives us the position of a point on a ray shot from the camera through the mid-sample of each pixel, at the closest triangle intersection. We call these positions *view samples* and they are the points for which we want to evaluate shadow (i.e. we want to evaluate whether these points are visible from the light source or not). We can do this by testing, for every view sample, whether it lies within the shadow frustum (i.e. the shadow volume of the triangle). If so, the sample is marked as shadowed in a separate buffer requiring a single bit per pixel. We will call this buffer the *shadow buffer* in the discussion below. To evaluate whether the sample is within the volume or not is a matter of testing the point against the four planes that make up the volume (See Figure 2).

While exhaustively testing all view samples against all triangle volumes is obviously not a good idea in practice, it is worthwhile to note a few things about this naive algorithm:

- It works without modification for any arbitrary triangle soup.
- It will be robust as long as some precautions are taken (see paragraph on robustness below)
- It requires a very small amount of extra memory storage (a single bit per view sample).
- It requires no pre-processing nor does it matter where the light, camera near or far planes are situated.
- When a pixel has been marked as in-shadow, it need no longer be considered by other triangle shadow volumes.

Let's consider what could be done to alleviate the overdraw problems in an imaginary customized stamp rasterizer with a two-level hierarchical depth buffer and an equally sized two-level hierarchical shadow buffer (which can be thought of as a one bit stencil buffer). Let's say the upper level of the hierarchical depth buffer contains the min and max of the 4x4 depth-values of the lower level. A texel's (x, y) coordinates and these two depths then define an axis aligned bounding box in normalized device coordinates (or a bounding-frustum in world space, see Figure 3), for the view samples contained within. We will refer to such bounding-frustums as *tiles* from here on.

Our imaginary rasterizer takes a triangle and a light-source position as input and rasterizes the projected shadow volume. The main difference in how our rasterizer works, compared to how shadow quads are traditionally rasterized, lies in the way that we cull against the hierarchical depth buffers. Where a traditional rasterizer will cull a number of fragments of a shadow quad only if they all lie in front of the min depth stored in the upper level of the hierarchy (for z-fail), our rasterizer tests the tile against each plane of the shadow frustum. If the tile is found to lie outside either plane, it can be culled and no bits will be set for the contained view samples in the shadow buffer. Additionally, if the tile is found to lie inside all planes, we know that all contained view samples are covered by this triangle, and so, we simply set a bit in the higher level of our hierarchical shadow buffer and can then safely abandon the tile. If the bounding-box can not be trivially rejected nor accepted, the individual view samples will be tested against the shadow volume planes and the lower level of the shadow buffer is updated.

Before the shadow buffer is used to determine whether a pixel should be considered in shadow or not, the two levels must be merged. This is done by, for each set bit in the higher level, also setting the corresponding 4x4 bits in the lower level, regardless of their current state.

For the algorithm to be perfectly robust, two things must be considered. First, if an edge is shared by two triangles, a view sample will be tested against the same plane twice, only with opposite normals. If the sample lies very close to the plane, we can get the erroneous result that the sample lies outside both, unless we make sure that the equations for these planes are constructed in exactly the same way, which may not happen if we simply use the vertices in the order they are submitted. Instead, when constructing a plane from the light's position and two edge vertices, the vertices are taken in an order defined by their world space coordinates (any unique ordering will do). Similarly, for a perfectly robust solution, testing whether a sample lies below the plane formed by the triangle should be done exactly in the same way as when the original depth buffer value was created. A hardware vendor could ensure that this is the case, but our software implementation must resort to adding a bias to the triangle plane to avoid self shadowing artifacts. Unlike the bias required for shadow maps, this bias can be very small and constant and in practice it does not introduce any noticeable artifacts.

3.1 A software hierarchical shadow-volume rasterizer

To evaluate the new algorithm, we have designed a hierarchical shadow volume rasterizer and implemented the design in software using NVIDIA's CUDA platform. We chose a fully hierarchical approach because this is the most viable known approach to parallel SIMD software rasterization [Abrash 2009], while for a hardware implementation it is likely that a two-level stamp rasterizer would prove more efficient.

The rasterizer extends the two-level approach presented earlier to be fully hierarchical, with $L = \lceil \log_s N \rceil$ levels, for some branching factor s and number of pixels, N, in both the hierarchical z-buffer and shadow buffer. These hierarchies represent an implicit full tree with branching factor s, which is traversed during rasterization of a triangle shadow volume.

Our design broadly follows the approach used for the 2D triangle rasterizer in Larrabee [Abrash 2009], which is illustrated in Fig-

Algorithm 1 Basic parallel traversal algorithm, for a square tile size $T \times T$, which assumes SIMD with $T \times T$ lanes. The algorithm is expressed as a program running on an individual SIMD lane, identified by *simdIndex* $\in [0..T \times T)$ and able to broadcast a single bit to each other using BALLOT. We use $\otimes$ to denote element-wise multiplication. Tiles are referenced using integer tuples defining their location within the current hierarchy level.

1: **procedure** TRAVERSAL(*level*, *parentTile*, *tri*)
2: *subTile* $\leftarrow$ (*simdIndex* $\bmod T$, *simdIndex*$/T$)
3: *tile* $\leftarrow$ *parentTile* $\otimes (T, T)$ + *subTile*
4: **if** *level* is final level **then**
5: **if** TESTVIEWSAMPLE(*tile*,*tri*) **then**
6: UPDATESHADOWBUFFER(*level*, *tile*)
7: **return**
8: *tileIntersects* $\leftarrow$ TESTTRIVOLUME(*level*, *tile*, *tri*)
9: **if** *tileIntersects* = *ACCEPT* **then**
10: UPDATESHADOWBUFFER(*level*, *tile*)
11: **else**
12: *queue* $\leftarrow$ BALLOT($tileIntersects$ = *REFINE*)
13: **for each** nonzero bit b_i **in** *queue* **do**
14: *child* $\leftarrow$ *parentTile* $\otimes (T, T) + (i \bmod T, i/T)$
15: TRAVERSAL(*level* + 1, *child*, *tri*)

ure 4, as this is simpler to illustrate and exactly analogous to what we do. The example shows 4×4 tiles, which would be suitable for 16-wide SIMD, with each lane processing a tile at the current level in the hierarchy. For each edge, a *trivial accept corner* and a *trivial reject corner* are found. These are the tile corners with greatest and least projection on the edge normal, as shown for tile 0 in the figure. If the trivial reject corner is outside *any* edge, then the tile can be rejected (shown in green). Conversely, if the trivial accept corner is inside *all* edges, then the tile can be trivially accepted (shown in blue). If neither of these conditions are satisfied (white), then the tile must be recursively refined at the next level of the hierarchy (Figure 4(b)). Note the single yellow tile shown (tile 9), which is obviously outside the triangle but cannot be rejected by the algorithm because its trivial reject corners are not outside any edge. This is sometimes called the *triangle shadow* [McCormack and McNamara 2000], and produces false positives unless some extra test is employed that is capable of rejecting such tiles.

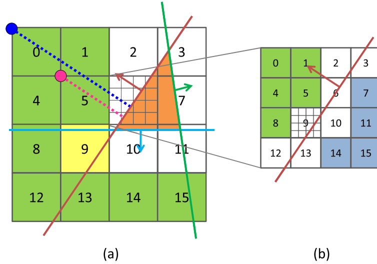


Figure 4: *Hierarchical 2D triangle rasterization illustrated by two levels. In (a), the green tiles are trivially rejected, white tiles need more refining and the yellow tile (9) is in the triangle shadow. The purple and blue dots show, for tile 0, the trivial reject and accept corners, respectively, to use with the red edge. In (b), showing the next level of refinement for tile 6, the blue tiles are trivially accepted.*

Our suggested process for rasterizing shadow frustums is very similar, with some notable differences. Firstly, as we are rasterizing shadow *frustums*, the three edge equations defining a triangle are replaced by four plane equations, which define the shadow frustum. Secondly, the hierarchical shadow buffer enables much simpler trivial accept handling; only a single bit needs to be updated in the correct hierarchy level. Lastly, our rendering is fully hierarchical instead of tiled, with each primitive traversing the hierarchy from the root and writing the results directly into the shadow-buffer hierarchy. Certain other implementation details are also different as we target an NVIDIA GPU rather than the Larrabee architecture. This is further elaborated on in Section 3.4.

Using homogeneous clip space is advantageous since, by preceding the perspective divide, it removes the need for clipping [Olano and Greer 1997]. The intersection test between a tile and triangle shadow volume is very similar to the frustum vs. AABB test – commonly used for view frustum culling – with the shadow volume being the frustum.

Traversing the hierarchy from the root, as opposed to using a tiled approach, has the advantages of improved scaling with resolution and that large shadow volumes can trivially accept or reject larger tiles. Even though triangles in today's complex scenes are often very small, the projected shadow frustums generated from such triangles can still be arbitrarily large, especially in the worst cases (see Figure 5). Efficiently handling the worst cases is important if we wish to construct a shadowing algorithm with low variability in frame times.

The basic SIMD traversal algorithm is shown in Algorithm 1. Each SIMD lane handles one sub-tile. They then exchange their results as bits in a single word, before recursively descending to the next level.

3.2 Textured and transparent shadows

As described in section 2, the original Shadow-Volume algorithm relies on extending shadow quads from the silhouette edges only. In that way, a large number of shadow quads can be culled away and overdraw is reduced, but we lose the ability to determine which triangles cover which view samples. If all we want is binary shadow information, this is acceptable, as a sample will be in shadow regardless of which or how many triangle shadow volumes it lies within. If one intends to draw textured shadows or shadows cast from semi-transparent triangles, however, all triangles that cover a view sample must be considered individually.

In our approach, the triangle shadow volumes are always considered individually. Hasselgren et. al. [2007] show that if all triangle shadow volumes are rendered separately, textured and semi-transparent shadows are feasible, but they do not suggest any method to render shadow frustums efficiently and so are limited to rather low polygon counts. We incorporate their ideas into our efficient rendering of shadow frustums and can render hundreds of thousands of textured or semi-transparent shadows in real-time. Below, we describe how textured and semi-transparent shadow-casters can be taken care of with very small changes to the original algorithm.

Semi-transparent shadow casters To incorporate semi transparent shadows in our method, we modify the hierarchical shadow buffer such that it contains a floating point value, instead of a bit, for every tile and view sample. The shadow buffer is cleared to zero. When updating the shadow buffer (UPDATESHADOWBUFFER in Algorithm 1), instead of setting a bit, $log(1 - \alpha)$ is atomically added. To merge the hierarchical shadow buffer into a single shadow buffer with a transmittance value per view sample, we simply add the value of a parent node to all its children instead of OR-

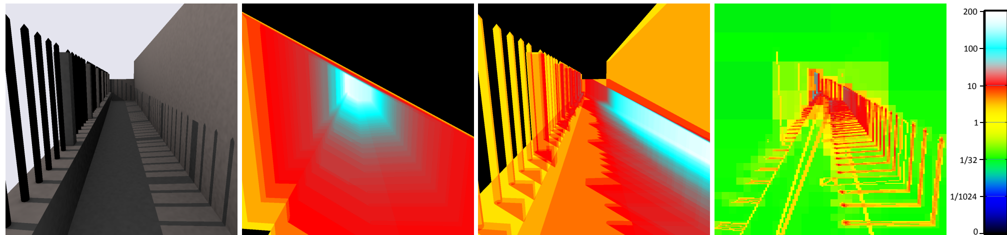


Figure 5: *Visualizing the overdraw caused by different algorithms (according to the metric given in Section 4). From left to right (total number of view sample test-and-set operations in parenthesis): The scene, z-pass algorithm (19.7 million), z-fail algorithm (18.3 million), ours (1.2 million). The overdraw in the first two algorithms is proportional to the sum of the areas of projected shadow-quads, whereas in our algorithm view-samples that lie outside the triangle shadow-frustums are quickly culled.*

ing them. The leaf nodes will now contain, for every view sample, $\sum log(1-\alpha_i) = log(\prod 1-\alpha_i)$ for every triangle i that lies between it and the light source. To get the transmittance, for each view sample we simply raise e to the power of that value. This allows us to efficiently render shadows from models with per-triangle alpha values, which is often sufficient to generate compelling images (see Figure 1(c)). If the alpha-value needs to be interpolated over the triangle or fetched from a texture, we can not trivially accept an internal node in this simple way, as explained in the next paragraph. Colored transparent shadows are trivially supported by applying the above scheme to each wavelength (e.g. to a shadow buffer of RGB-tuples).

Textured shadow casters Computing interpolated texture coordinates to support textured shadow casters is surprisingly simple in our method. Recall from section 3.1 that to determine whether a view sample is in shadow, we test its position against the four planes that make up the shadow frustum and evaluate the sign of the results. Though perhaps not immediately obvious, it can be shown that the distances obtained from each of the planes generated from the triangle edges (d_0, d_1, d_2), are indeed a scaled version of the barycentric coordinates on the triangle. Scaling these distances such that $d'_0 + d'_1 + d'_2 = 1$, we have the true barycentric coordinates and can obtain the texture coordinates for the view sample. Moreover, this same approach holds in clip space, so no transformations are required. Given the texture coordinates, we can get the alpha-mask value, opacity value, or colored opacity value from a texture and proceed to update the shadow buffer for a view sample.

Note that while it is simple for us to introduce support for textured shadow casters, we are forced to abandon the trivial accept optimization described in section 3.1. It is simply not the case any more that if a tile lies entirely within the shadow volume of a triangle, all sub-tiles will have the same shadow value. Indeed, the triangle may cast no shadow if its texture is empty. Instead, when a tile is trivially accepted, we flag it as such and immediately traverse all sub-tiles without testing, until we reach the individual samples, for which we evaluate the texture and update the final level of the shadow buffer. This gives worse performance, of course, than when trivial accept is as simple as updating a shadow-buffer node, but still works at acceptable frame rates for complex models. There is a large drop in performance in our implementation, however, when the shadow of a single or very few triangles cover a large part of the screen. In this case, only one or a few multiprocessors will have any work to do and load-balancing becomes a problem. To alleviate this, we could instead employ the method described in [Abrash 2009] to trivially accept a tile. The tile and shadow frustum pair would then simply be pushed to a work queue that could be processed efficiently in a separate pass.

As noted previously, alpha-masked shadow casters are trivially supported by the shadow-mapping algorithm. When rendering the shadow map, a fragment can simply be discarded if the alpha value is below some threshold. Real valued alpha textures, however, are not easily supported, and one has to use more complex shadow-mapping techniques for this to work (e.g. Stochastic Transparency [Enderton et al. 2010]). Even when rendering alpha-masked shadow casters, the projection of a fragment onto the shadow map rarely covers a single texel, and so, filtering should be employed, and then the simple alpha-mask texture again returns a real valued result which will be thresholded. Our method trivially handles filtered lookups into an alpha-mask texture, and consequently, produces higher quality shadows (this too was noted by [Hasselgren and Akenine-Moller 2007]).

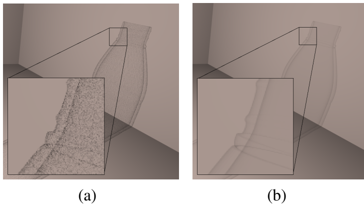
Algorithm 2 Testing a shadow frustum against a tile. The algorithm first constructs the normalized device coordinate representation of the tile, and then tests the *trivial-reject* and *trivial-accept* corners against each of the four planes that define the shadow frustum. The xy extents of the tiles at a level, in normalized device coordinates, are available through the constant *tileSize*$_{level}$.

```
 1: procedure TESTTRIVOLUME(level, tile, tri)
 2:     tileMin.xy ← (−1.0, −1.0) + tile ⊗ tileSize_level
 3:     tileMin.z ← fetchMinDepth(level, tile)
 4:     tileMax.xy ← tileMin.xy + tileSize_level
 5:     tileMax.z ← fetchMaxDepth(level, tile)
 6:     numInside ← 0
 7:     for each plane p_i in tri do
 8:         if TESTPLANEAABB(p_i, tileMin, tileMax) > 0 then
 9:             return REJECT
10:         else
11:             numInside ← numInside + 1
12:     if numInside = 4 then
13:         return ACCEPT
14:     else
15:         return REFINE
```

Stochastic transparent shadows When the shadow buffer contains a float per node instead of a single bit, the memory requirements are obviously much higher. Especially for high quality antialiased render targets (MSAA, CSAA or SSAA buffers) where every pixel has several depth samples, each of which should be tested against the shadow volumes for correct shadows, the memory footprint may be a limiting factor to the usefulness of our algorithm (or any other sample-accurate transparent shadows algorithm). For ex-

ample, a 1920x1080 buffer with 16 depth samples per pixel and 32-bit float transmittance values would require 130MB of memory for the final level. Therefore, we suggest a different approach, where every pixel sample still holds a single bit of shadowing information. When updating the shadow buffer with a semi transparent shadow caster, the bit is simply set stochastically with a probability equal to α. The shadow buffer is used as per usual to decide whether each sample shall be considered lit or in shadow. When resolving the final pixel color the result will be noisy but correct on average (see Figure 6(a)). The proofs to why this works are equivalent to those in [Enderton et al. 2010], and, while we have not implemented it, the same scheme to stratify samples over a pixel as is presented in that paper should work well to reduce the noise.

Transparent shadow receivers We have shown that semi-transparent shadow casters present no problems to our algorithm. Since it is, like the original shadow-volume algorithm, essentially a *deferred* rendering algorithm, transparent shadow receivers are not quite as trivial, though. Since the light-visibility calculations do not (as with e.g. shadow maps) happen during fragment shading, but in a post-processing pass, simple techniques where polygons are sorted on depth before rendering will not work with our algorithm. Oftentimes, these approaches are not sufficient anyways, as they are prone to errors (two triangles may span the same depth and cannot be uniquely sorted). Our algorithm works well with depth-peeling [Everitt 2001], where layers of transparent objects are rendered in several passes and with Stochastic Transparency [Enderton et al. 2010], although the latter produces z-buffers with a high variance which causes a hierarchical z-buffer to be less than optimal.



(a) (b)

Figure 6: *Stochastic (a) and Real valued (b) transparent shadows.*

3.3 Antialiasing

Hard shadow edges often mean that two neighboring pixels will have vastly different intensities, and so, anti-aliasing can greatly improve image quality. Our algorithm works well with full screen anti-aliasing schemes like MSAA, CSAA or SSAA as long as the pixel-sample positions' can be obtained. For the shadow calculations, such a buffer is simply considered a large render target, and when the shadow buffer has to be updated for a view sample, the pixel sample positions offset is fetched from a table.

We also suggest a novel anti-aliasing scheme that requires no extra depth-samples per pixel. When the scene is rendered from the camera, an additional color buffer is rendered that contains the x and y derivatives of the fragment's depth. Building the depth buffer hierarchy works exactly as before, except the derivatives are used to find a minimum and a maximum depth already at the lowest level. The shadow buffer hierarchy is allocated with one additional level (so that a view sample will have a number of shadow bits instead of one single bit), and when traversing the shadow frustums through the hierarchical depth buffer, we simply traverse as though there were an additional level of the depth hierarchy, but the final view samples to be tested are generated from the samples (x, y) positions in the pixel and the depth derivatives. The final shadow value used for the pixel will be the ratio of set bits to clear bits in the shadow buffer. Note that this is equivalent to projecting a fragment on a shadow map of infinite resolution and taking a number of PCF taps within this region.

3.4 Implementation

We have implemented our algorithm using CUDA, where the native SIMD group (called a *warp*) is 32 threads wide. One warp is issued per shadow frustum (with enough warps in each CUDA block to fully utilize the hardware), and the threads in each warp cooperate in rasterization of the frustum. The threads in a warp can efficiently exchange bit flags using the `__ballot` intrinsic (available on NVIDIA GPUs of compute capability 2.0 and above). On devices that lack this instruction, a parallel reduction in shared memory will yield the same result, at some cost in performance. Choosing a branching factor that matches the SIMD width allows the traversal to entirely avoid divergence (threads within a warp executing different code paths, for example if shadow frustums traverse the tree to different depths). A branching factor of 32 also matches the 32-bit word width, which makes updating bit masks simple and efficient.

However, 32 items cannot tile a square region. To construct a tree from a square frame buffer, we instead alternate between 8×4 and 4×8 at each level. The implementation is otherwise faithful to the traversal algorithm (Algorithm 1) presented earlier. To accumulate results in the shadow-buffer hierarchy, we use atomic operations, e.g. the `atomicOr` intrinsic. While atomic operations are often held to be slow, we were not able to observe any penalty from using them, which may be because of the relatively low load the depth first traversal places on the memory subsystem.

In order to support transparency, we need to use 32 floating point numbers per tile instead of 32 bits used for binary shadow. Updating these is done by using atomic add from each SIMD lane. To handle colored transparent shadows, we simply use three atomic adds, one per component.

3.5 Optimizations

Culling against shadow frustum silhouette As described in section 3.1, our shadow frustum vs. tile test is conservative, and can thus produce false positives leading to tiles refined unnecessarily. This problem is the 3D equivalent of the triangle-shadow problem for 2D rasterization (see Section 3.1), and causes traversal to refine tiles that are outside the projected shadow volume. To improve culling efficiency, we also test two additional edges that define the 2D projection of the shadow volume (illustrated using red lines in Figure 2). The new edges are defined in 2D homogeneous clip space, and are tested in a very similar fashion to the planes already used. Adding this test helps ensuring that we do not visit any tiles not actually within the on-screen shadow.

Front face culling When rendering closed objects, we can choose to use only the triangles that face the light *or* the back-facing triangles as shadow casters [Zioma 2003]. This is also employed in shadow mapping, where rendering only back facing triangles can reduce self shadowing artifacts. For our algorithm, there is an even more compelling reason to use this approach. Consider what happens when a front-facing triangle that is visible from the camera is used as a shadow caster. This triangle will have to traverse the hierarchical depth buffer all the way down to the view samples that belong to that triangle, since all of these will lie exactly on the triangle plane. Unlike the traditional shadow-volume algorithm, using

this optimization does not require that the model is actually modeled as a two manifold mesh. As long as the object will render correctly to screen with backface-culling, it will work robustly as a shadow caster with front-face culling. For example, unclosed backdrop geometry will cast shadows properly.

Remove unlit depth samples We want to avoid computing light visibility for view samples that are already unlit, either because of the sample not facing the light, or being unlit for other reasons like being part of the background. To achieve this, we flag unlit view samples and do not include their depths when building the hierarchical depth buffers.

Maintaining an updated shadow buffer When traversing all shadow frustums through the hierarchical depth buffer, we set bits in the hierarchical shadow buffer representing completely shadowed tiles or view samples. Naturally, once a tile or view sample is found to be in shadow, given some triangle frustum, its state cannot change. Therefore an obvious optimization to our traversal algorithm is to stop traversal as soon as we reach a node that is already marked as being in shadow. It is simple to modify Algorithm 1 to AND the bitmask *queue* with the inverse of the current shadow buffer value for the node. This, however, will only stop the shadow frustum from being traversed through nodes that have previously been trivially accepted by some shadow frustum (not tiles that have been filled by several different shadow frustums), so the gain in efficiency is modest. Alternatively, a thread that fills a node can either recursively propagate that change up the hierarchy or simply update the one level above and rely on the changes to propagate due to other threads over time. Neither method improves performance in our implementation however, probably due to the increased cost of reading the shadow-buffer and the potentially long latency before an update is visible to other threads. An even more efficient optimization would be to keep the hierarchical depth buffer dynamically updated (by removing shadowed tiles from the hierarchy), but this seems less likely to be feasible.

4 Results and Discussion

To evaluate the performance of our algorithm, we have implemented carefully tuned versions of the z-pass and z-fail algorithms. Our implementations are similar to those suggested by [Aldridge and Woods 2004], except they are implemented using shaders and run entirely on the GPU. Since the stencil buffer can only be incremented or decremented by one, shadow quads shared by two triangles are rasterized twice, as this is much faster than replacing the stencil buffer with a color buffer (which, using blending, can be incremented by two or more). In the z-fail implementation, we render the near and far caps, while in the z-pass we need only render the shadow quads. Both implementations rely on depth clamping to avoid clipping artifacts. In the z-pass algorithm, we initialize the stencil buffer with a value corresponding to the number of shadow volumes the camera is in. As mentioned, establishing this value robustly is still an unsolved problem and occasionally causes grave artifacts. Both z-pass and z-fail can also fail if the eight-bit stencil buffer overflows.

The time taken to render the shadow volumes is plotted for a fly-through of a game-scene (see supplementary video), in Figure 7. We show results for two resolutions, 1024x1024 and 4096x4096. This latter resolution may seem extravagant, but really corresponds to approximately the same number of samples that would be processed for an image rendered in 1080p with 8xMSAA. The timings reported for our algorithm are those measured for generating the shadow buffer (build depth hierarchy, triangle-setup, rasterization and final merging) and omit any redundant buffer copies that happen when mixing OpenGL and CUDA. The timings presented for z-pass omit the time taken to evaluate how many shadow casters lie between the light and the camera. The graphs show the performance with and without the front-face culling (FFC) optimization described in section 3.5. The scene used is a part of the freely available *Epic Citadel* [Epic Games 2011] ($\sim$ 60k triangles) which contains many open edges, in what are really closed objects, and so would not have worked with the simpler shadow-volume algorithms. The scene has been slightly modified to contain no one-sided geometry (the cloth in the original model). All timings were measured on an NVIDIA GTX480 GPU.

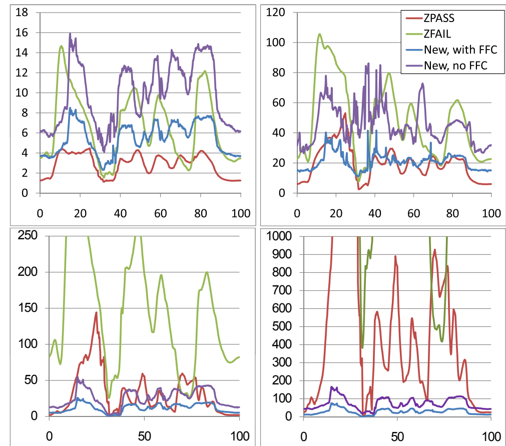


Figure 7: *Comparing z-pass, z-fail and the new algorithm, with and without* Front Face Culling *(FFC), in a fly-through of the citadel scene. Above: Time taken (in ms) to generate per–view-sample shadow information. Below: Millions of test-and-set operations required. Left is for a render target of size 1024x1024, right 4096x4096.*

As can be seen from Figure 7, when front-face culling is enabled, our software GPU implementation outperforms z-fail even at the lower resolution, and performs with a similar average as z-pass at the higher resolution, though with much lower variability. Without front-face culling, z-pass is still faster than our algorithm at the higher resolution but, as noted previously, the z-pass algorithm is not entirely robust and so our algorithm is a compelling alternative. Moreover, for meshes without connectivity information or with short silhouette loops (e.g. destructible buildings, or a flock of birds) our frame times stay low while z-pass rendering times would increase significantly.

To further examine the performance of our algorithm we have measured the total number of test-and-set operations per view sample. For the shadow-volume algorithms, we have ignored the work required by the rasterizer and early z-culling (as we lack information to properly evaluate that), and so the number of test-and-set operations reported is simply the total number of stencil updates required for a frame. For our algorithms, we have counted the total number of tile/shadow frustum and sample/shadow frustum tests performed.

Figure 7 shows the number of test-and-set operations for the same scene and animation as before. Clearly, the new hierarchical algorithm is more efficient, even at relatively low resolutions, while at the high resolution it is especially effective. As expected, our

hierarchical approach scales well with increasing resolutions: raising the number of view samples sixteenfold only requires between about two to four times as many test-and-set operations, while for z-pass the increase is around 16 times. The results also demonstrate the low variability of the new algorithm which is due to our algorithms ability to trivially accept large tiles that lie within the shadow frustum and to trivially reject tiles that lie outside. Observe that, when front-face culling is disabled, the number of required test-and-set operations is more than doubled (roughly 2.2 times for the lower, and around 2.4 times for the higher resolution). This is the expected behavior, as visible front faces must be refined all the way to the sample level (see Section 3.5), and shows that front-face culling ought to be enabled whenever possible.

Another important consideration for a shadowing algorithm is robustness and the artifacts it may produce. The z-pass algorithm is generally not robust, because of the camera in shadow problem, and because of stencil buffer overflow, which also affects the z-fail algorithm. When these failures are encountered, the shadow computed for the *entire* screen may be incorrect – a highly disturbing artifact. Our algorithm, on the other hand, has no inherent robustness issues, and will at worst produce light leakage if the mesh is not properly welded.

The amount of memory required by our basic (non transparent) algorithm, is very low. For a 4096x4096 rendertarget our hierarchical shadow buffer requires only 2.1MB, besides the resident depth buffer, our hierarchical depth buffers require an additional 2.1MB for a total of 4.2MB. An eight-bit stencil buffer, which is a bare minimum for shadow-volume algorithms requires $\sim$ 16MB for the same resolution. The alias-free shadow-map implementation described in [Sintorn et al. 2008] stores all view sample positions (three floats) in a compact array of lists per light space pixel which would take $\sim$ 200MB at this resolution. Additionally, they require a shadow map where each texel needs to store as many bits as are the maximum list size. For a shadow map of 1024x1024 and a max list size of 512 that would mean an additional 16MB of memory. For comparison, a 4096x4096 omnidirectional shadow map would require 384MB.

The results of the stochastic shadow buffer described in section 3.2 are demonstrated in Figure 6. The images are rendered at a resolution of 4096x4096 and downsampled to 1024x1024. Again, this is to illustrate how the algorithm could work with images rendered using high quality MSAA or CSAA. While it is quite noisy, the stochastic image renders slightly faster than using a float value per sample and, more importantly, requires only a single bit of visibility information per view sample. The memory footprint of our algorithm is thus reduced from the original $\sim$ 64MB in Figure 6(b), to only $\sim$ 2MB in Figure 6(a).

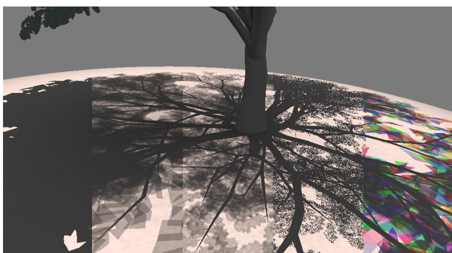


Figure 8: *The results of rendering a scene with different variants of our algorithm.*

Figure 8 shows the tree from figure 1(b) rendered with different variants of our algorithm. The algorithm and the time taken to generate the shadows were (from left to right): standard binary visibility (7.9ms), semi-transparent shadow casters where all leafs have constant $\alpha = 0.5$ (8.6ms), semi-transparent textured shadow casters (12.9ms), stochastic semi-transparent textured shadow casters (12.9ms), and colored semi transparent shadow casters (10.4ms).

The choice of 32 as a branching factor for the hierarchical depth and shadow buffers is natural, as this matches both native word size and SIMD width. However, a high branching factor results in more wasted work; for example, all shadow frustums that are not culled in the setup phase will need to test all 32 tiles in the first level of the hierarchy, whereas a binary tree would only need to test two. Lower branching factor, on the other hand, results in deeper trees and more divergence. We have not explored this trade-off.

5 Future Work

The optimization where unlit samples are removed enables the use of a two-pass approach, where a first pass runs the algorithm only on those triangles that are expected to be good blockers by some heuristic, then refreshes the hierarchy by removing yet more unlit samples, and finally traverses the remaining triangles [Olsson and Assarsson 2011]. As constructing the depth hierarchy is cheap, this optimization may yield a significant increase in efficiency, especially if *blocker geometry* (i.e. conservative simple geometry) is placed manually or generated.

The algorithm could be extended to handle soft shadows quite simply, in a manner similar to that of [Sintorn et al. 2008]. The triangle frustums would then be expanded to include the whole influence region of the triangle, given an area light source, and the shadow buffer could contain, for each view sample, a bit per light sample. Also, our novel antialiasing scheme resembles PCF filtering in many ways. It seems likely that this algorithm could be modified to support samples taken outside the pixels' bounding box, to support PCF style blurred shadows in our algorithm. Several problems remain to be solved in these areas, though.

6 Conclusion

We have presented a novel shadow algorithm based on individual triangle shadow volumes, which combines many of the strengths of shadow maps and shadow volumes. We also demonstrated a GPU software implementation of a hierarchical rasterizer that supports the algorithm. Despite running entirely in software, it competes well against highly tuned implementations of shadow volumes which rely heavily on hardware acceleration, and offers realtime performance. Meanwhile, the new algorithm is completely robust, works for any aribtrary collection of triangles and integrates easily into a deferred rendering pipeline making it a compelling choice for rendering pixel accurate shadows, especially at high resolutions.

References

ABRASH, M. 2009. Rasterization on larrabee. *Dr. Dobbs Journal*.

AILA, T., AND AKENINE-MÖLLER, T. 2004. A hierarchical shadow volume algorithm. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, HWWS '04, 15–23.

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proc. of EGSR 2004*, 161–166.

ALDRIDGE, G., AND WOODS, E. 2004. Robust, geometry-independent shadow volumes. In *Proc. of 2nd international conf.*

on Computer graphics and interactive techniques in Australasia and South East Asia, GRAPHITE '04, 250–253.

ANNEN, T., MERTENS, T., SEIDEL, H.-P., FLERACKERS, E., AND KAUTZ, J. 2008. Exponential shadow maps. In *Proc. of graphics interface 2008*, GI '08, 155–161.

ARVO, J. 2004. Tiled shadow maps. In *Proc. of Computer Graphics International 2004*, 240–246.

BILODEAU, W., AND SONGY, M., 1999. Real time shadows. Creativity 1999, Creative Labs Inc. Sponsored game developer conferences, Los Angeles, California, and Surrey, England.

CARMACK, J., 2000. Z-fail shadow volumes. Internet Forum.

CARPENTER, L. 1984. The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph. 18* (January), 103–108.

CHAN, E., AND DURAND, F. 2004. An efficient hybrid shadow rendering algorithm. In *Proc. of the EGSR*, 185–195.

CROW, F. C. 1977. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11* (July), 242–248.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *Proc. of i3D 2006*, I3D '06, 161–165.

EISEMANN, E., ASSARSSON, U., SCHWARZ, M., AND WIMMER, M. 2009. Casting shadows in real time. In *ACM SIGGRAPH Asia 2009 Courses*, SIGGRAPH Asia 2009.

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2010. Stochastic transparency. *IEEE TVCG 99*.

EPIC GAMES, 2011. Unreal development kit: Epic citadel. http://www.udk.com/showcase-epic-citadel.

EVERITT, C., AND KILGARD, M. J., 2002. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Published online at http://developer.nvidia.com.

EVERITT, C., 2001. Interactive order-independent transparency. Published online at http://www.nvidia.com/object/Interactive_Order_Transparency.html.

FERNANDO, R. 2005. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH 2005.

FOREST, V., BARTHE, L., GUENNEBAUD, G., AND PAULIN, M. 2009. Soft textured shadow volume. *Computer Graphics Forum, EGSR 2009 28*, 4, 1111–1121.

FUCHS, H., GOLDFEATHER, J., HULTQUIST, J. P., SPACH, S., AUSTIN, J. D., BROOKS, JR., F. P., EYLES, J. G., AND POULTON, J. 1985. Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes. *SIGGRAPH Comput. Graph. 19* (July), 111–120.

HASSELGREN, J., AND AKENINE-MOLLER, T. 2007. Textured shadow volumes. *Journal of Graphics Tools*, 59–72.

HEIDMANN, T. 1991. Real shadows, real time. *Iris Universe 18*, 28–31. Silicon Graphics, Inc.

HORNUS, S., HOBEROCK, J., LEFEBVRE, S., AND HART, J. C. 2005. ZP+: correct Z-pass stencil shadows. In *ACM symp. on Inter. 3D Graphics and Games, I3D, April, 2005*, 195–202.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. on Graphics 24*, 4, 1462–1482.

KIM, T.-Y., AND NEUMANN, U. 2001. Opacity shadow maps. In *Proc. EG Workshop on Rendering Techniques*, 177–182.

KIM, B., KIM, K., AND TURK, G. 2008. A shadow-volume algorithm for opaque and transparent nonmanifold casters. *journal of graphics, gpu, and game tools 13*, 3, 1–14.

LAINE, S. 2005. Split-plane shadow volumes. In *Proc. of Graphics Hardware 2005*, 23–32.

LAURITZEN, A., SALVI, M., AND LEFOHN, A. 2011. Sample distribution shadow maps. In *Proc.*, I3D '11, 97–102.

LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution matched shadow maps. *ACM TOG 26*, 4, 20:1–20:17.

LLOYD, B., WEND, J., GOVINDARAJU, N. K., AND MANOCHA, D. 2004. Cc shadow volumes. In *EGSR/Eurographics Workshop on Rendering Techniques*, 197–206.

LLOYD, D. B., GOVINDARAJU, N. K., QUAMMEN, C., MOLNAR, S. E., AND MANOCHA, D. 2008. Logarithmic perspective shadow maps. *ACM TOG 27* (November), 106:1–106:32.

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. In *Proc. SIGGRAPH 2000 (Aug.)*, SIGGRAPH 2000, 385–392.

MCCORMACK, J., AND MCNAMARA, R. 2000. Tiled polygon traversal using half-plane edge functions. In *Proc. of ACM workshop on Graphics hardware*, HWWS '00, 15–21.

MCGUIRE, M., AND ENDERTON, E. 2011. Colored stochastic shadow maps. In *Proc. of i3D'11 (Februari.)*.

OLANO, M., AND GREER, T. 1997. Triangle scan conversion using 2d homogeneous coordinates. In *Proc. of ACM workshop on Graphics hardware*, 89–95.

OLSSON, O., AND ASSARSSON, U. 2011. Improved ray hierarchy alias free shadows. Technical Report 2011:09, Chalmers University of Technology, may.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering antialiased shadows with depth maps. In *Proc.*, SIGGRAPH 87, 283–291.

SINTORN, E., AND ASSARSSON, U. 2009. Hair self shadowing and transparency depth ordering using occupancy maps. In *Proc.*, i3D '09, 67–74.

SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample-based visibility for soft shadows using alias-free shadow maps. *CG Forum (EGSR 2008) 27*, 4 (June), 1285–1292.

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective shadow maps. In *Proc.*, SIGGRAPH 2002, 557–562.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12* (August), 270–274.

WIMMER, M., SCHERZER, D., AND PURGATHOFER, W. 2004. Light space perspective shadow maps. In *Rendering Techniques 2004 (Proc. EGSR)*, 143–151.

YUKSEL, C., AND KEYSER, J. 2008. Deep opacity maps. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2008) 27*, 2.

ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *Proc. of the 2006 ACM international conf. on Virtual reality continuum and its applications*, VRCIA '06, 311–318.

ZIOMA, R. 2003. Reverse extruded shadow volumes. In *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9*, W. Engel, Ed. Wordware Publishing, 587–593.

Paper V: Per-Triangle Shadow Volumes Using a View-Sample Cluster Hierarchy

Erik Sintorn, Viktor Kämpe, Ola Olsson and Ulf Assarsson

Abstract: Rendering pixel-accurate shadows in scenes lit by a point light-source in real time is still a challenging problem. For scenes of moderate complexity, algorithms based on Shadow Volumes are by far the most efficient in most cases, but traditionally, these algorithms struggle with views where the volumes generate a very high depth complexity. Recently, a method was suggested that alleviates this problem by testing each individual triangle shadow volume against a hierarchical depth map, allowing volumes that are in front of, or behind, the rendered view samples to be efficiently culled. In this paper, we show that this algorithm can be greatly improved by building a full 3D acceleration structure over the view samples and testing per-triangle shadow volumes against that. We show that our algorithm can elegantly maintain high frame-rates even for views with very high-frequency depth-buffers where previous algorithms perform poorly. Our algorithm also performs better than previous work in general, making it, to the best of our knowledge, the fastest pixel-accurate shadow algorithm to date. It can be used with any arbitrary polygon soup as input, with no restrictions on geometry or required pre-processing, and trivially supports transparent and textured shadow-casters.

I3D '14: Proceedings of the 2014 symposium on Interactive 3D graphics and games, to appear, March, 2014

Per-Triangle Shadow Volumes Using a View-Sample Cluster Hierarchy

Erik Sintorn* Viktor Kämpe* Ola Olsson* Ulf Assarsson*

Chalmers University of Technology

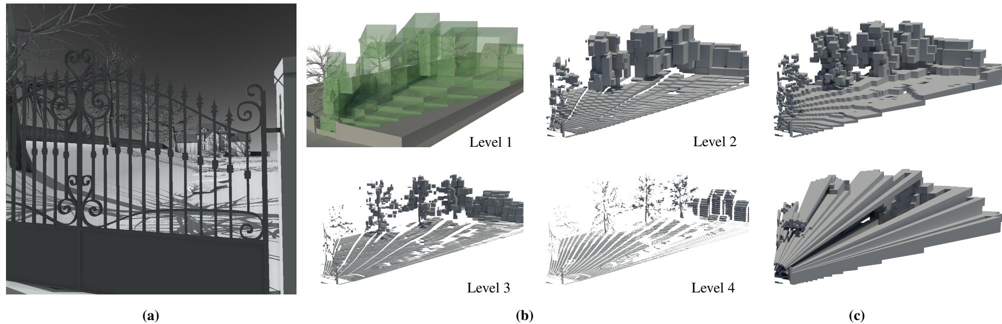


Figure 1: *a) A challenging scene for most real-time shadow algorithms, rendered in 4.2ms with our cluster hierarchy, 7.46ms with the original PTSV, and 12.16ms with ZPASS. b) The final four levels of our view-sample acceleration structure visualized. c) Top: Level 3 without explicit bounds. Bottom: The corresponding level when using the original PTSV algorithm.*

Abstract

Rendering pixel-accurate shadows in scenes lit by a point light-source in real time is still a challenging problem. For scenes of moderate complexity, algorithms based on Shadow Volumes are by far the most efficient in most cases, but traditionally, these algorithms struggle with views where the volumes generate a very high depth complexity. Recently, a method was suggested that alleviates this problem by testing each individual triangle shadow volume against a hierarchical depth map, allowing volumes that are in front of, or behind, the rendered view samples to be efficiently culled. In this paper, we show that this algorithm can be greatly improved by building a full 3D acceleration structure over the view samples and testing per-triangle shadow volumes against that. We show that our algorithm can elegantly maintain high frame-rates even for views with very high-frequency depth-buffers where previous algorithms perform poorly. Our algorithm also performs better than previous work in general, making it, to the best of our knowledge, the fastest pixel-accurate shadow algorithm to date. It can be used with any arbitrary polygon soup as input, with no restrictions on geometry or required pre-processing, and trivially supports transparent and textured shadow-casters.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing and texture;

Keywords: shadows, alias-free, real-time

*e-mail:erik.sintorn|kampe|olaolss|uffe@chalmers.se

1 Introduction and Previous Work

In this paper, we suggest a novel method for rendering pixel-accurate shadows from point-light sources in real time. While an abundance of very fast shadow algorithms are available (see, e.g., Real Time Shadows [Eisemann et al. 2011] for a recent overview of shadow algorithms in general), the vast majority are image-based approximative approaches, based on Shadow Mapping [Williams 1978]. In these algorithms, a discrete image (a shadow map) that contains point samples of the closest distance to the shadow casters from the light is generated. This shadow map is then queried while shading to determine whether an arbitrary point is in shadow or not. As the shadow map is generated without taking the actual points to be shaded into account, the result of the query can only be approximate and has to be filtered to reduce aliasing artifacts. To avoid having to use too large shadow maps and too large filter kernels, it is common practice to partition the view frustrum and use different shadow-maps for each partition (e.g. [Zhang et al. 2006]). With proper filtering and a good partitioning scheme, it is possible to obtain very high quality shadows that may be sufficient for many real-time applications, such as video games, but pixel-perfect shadows cannot be guaranteed. With virtual texturing, extremely high resolutions are possible, however [Lefohn et al. 2007], at the cost of view-dependent performance and memory requirements.

A second class of algorithms are those where the actual view samples to be considered are first collected and organized in some form of acceleration structure [Aila and Laine 2004; Johnson et al. 2005]. The shadow-casting polygons are then rendered over these irregular samples to determine light visibility for each view sample. While GPU-based implementations of these algorithms exist (e.g. [Sintorn et al. 2008]), the irregular rasterization process often leads to very unbalanced workloads, which results in uneven and potentially very poor performance.

A third class of algorithms are those based on Shadow Volumes [Crow 1977]. Here, for each shadow-casting object, a polygonal mesh (the shadow volume) that encloses the space that is blocked from the light by that object is generated. The shadow volumes are

then tested against all view samples, and a view sample is considered in shadow if it lies within any of the volumes. Shadow volumes were for a while frequently used in practice after a version of the original shadow-volume algorithm was introduced in which these inclusion tests could be performed efficiently on graphics hardware [Heidmann 1991]. We will refer to this algorithm as ZPASS in the remainder of this text. The basic idea is to extract all *sillhouette edges* from the mesh, each frame, and extrude these infinitely far away from the light source to form a *shadow quad*. The scene is first rendered from the camera's point of view into a depth buffer. Then, the shadow quads are rasterized as polygons onto a cleared stencil buffer while testing against this depth buffer. The stencil buffer is incremented for every front-facing polygon and decremented for every back-facing polygon. The resulting stencil value will be zero only when the view sample does not lie within any shadow volume. This algorithm only works well as long as the camera itself is not inside a shadow volume. To avoid that problem, the ZFAIL algorithm was introduced [Carmack 2000; Bilodeau and Songy 1999]. The only difference here is that the standard depth test is reversed so that all shadow quads that lie *behind* the z-buffer are rendered instead. This algorithm is more robust, but typically slower due to a higher fill rate.

The main problem with the traditional shadow volume algorithms lie in that they cull view samples only on their two-dimensional position in view space. Thus, a shadow quad must be tested against the potentially very large number of samples that lie within the volume formed by the quad and the camera position. Sintorn et al. [2011] alleviate this significantly by building a min-max hierarchy over the depth buffer and testing individual triangle shadow volumes against this hierarchy. A shadow volume can then be culled as long as it does not intersect the frustum formed by a node (or if the entire node is within the shadow volume). The authors show that the number of actual test-and-set operations required are dramatically reduced and that performance of their implementation is on par with, or better than, previous algorithms. Additionally, since each view sample can be tested against each triangle shadow volume, this algorithm trivially supports textured and semi-transparent shadow casters, and it can robustly handle any arbitrary set of shadow-casting polygons, without connectivity information. We will refer to this algorithm as *Per Triangle Shadow Volumes* (PTSV) in the remainder of this paper.

There are several other papers that attempt to reduce the fill-rate problems inherent in the traditional shadow-volume algorithm. In the work by LLoyd et al. [2004], the shadow volumes are culled and clamped per object in the scene graph, to reduce unnecessary overdraw. These methods are orthogonal to our algorithm. Aila and Akenine Möller [2004] identify tiles that lie on the shadow boundary and need to perform full per pixel tests only for these tiles. Chan and Durand [2004] attempt to find umbra regions and identify shadow boundaries using a shadow map before reverting to standard shadow volumes for only the pixels that lie on these boundaries. Finally, in Split Plane Shadow Volumes [Laine 2005], the number of stencil updates are reduced by locally (per tile) choosing whether to use the ZPASS or ZFAIL algorithm.

In this paper, we improve on PTSV by building a complete three-dimensional acceleration structure over the view samples, allowing *clusters* of samples that have the same two-dimensional bounds to be considered separately when they lie at different depths. Figure 1 illustrates a case where all previous work will perform very badly. A house in the far distance is viewed through a nearby gate, and a number of trees cast complex shadows that intersect the volume in between. The ZPASS algorithm [Heidmann 1991] will need to consider all view samples that have a depth which is further away than the shadow quad, i.e., essentially all view samples that do not lie on the gate. The ZFAIL algorithm [Bilodeau and Songy 1999; Carmack 2000], in contrast, must consider all samples that lie in *front* of the shadow quad. Unfortunately, the PTSV algorithm cannot do much better in this case, since most 8×4 tiles in the image will contain view samples from both the foreground and the background (see Figure 1c).

A very similar problem exists in the realm of real-time shading with many lights, where tiled shading algorithms have recently become popular [Olsson and Assarsson 2011; Harada 2012]. Similarly to PTSV, tiled-shading algorithms are sensitive to depth discontinuities. Our proposed algorithm is inspired by a recent solution to that problem, called *Clustered Shading* [Olsson et al. 2012]. The problem in tiled shading is analogous to that of PTSV, with the difference that the volumes considered are not shadow volumes but the bounding volumes of lights with a finite range (as is common in real-time applications). Olsson et al. observe that depth discontinuities can lead to many false positives where light volumes intersect tile volumes but none of the samples within, and that this problem is highly view dependent, leading to high variability in rendering times. They show that by clustering samples into three-dimensional subdivisions, as opposed to two dimensions for tiled, light-culling efficiency becomes much higher and view dependence lower, especially when considering many small light sources.

The main contributions in this paper are:

- A better view-sample acceleration structure for PTSV, which has a much smaller total volume and improves efficiency and performance significantly, making our algorithm the fastest real-time alias-free shadow method to date.
- A two-pass algorithm which removes performance spikes that are due to poor load balancing, at no extra cost.
- An improved set of culling planes over PTSV, which significantly reduces the number of false positives during traversal.

Meanwhile, our algorithm still maintains all the good properties of the PTSV algorithm. Shadow casters can be any arbitrary triangle soup with no additional connectivity information and we also inherit the ability to trivially support textured or semi-transparent shadow casters.

2 Algorithm

The goal of our algorithm is to establish, for every view sample in the G-buffer, whether that view sample is directly visible by a point light source or not. Samples that are blocked from the light source will be in shadow and the rest will have direct lighting applied in a final shading pass. We accomplish this by generating an acceleration structure over the view samples and then testing the shadow volume of each triangle against this structure. This approach has been attempted several times before (e.g. [Aila and Laine 2004; Sintorn et al. 2008; Sintorn et al. 2011]) but in this paper we will suggest that the quality of the acceleration structure is critical to obtaining good and reliable performance, and so we will generate a tightly fitting, fully three-dimensional hierarchy.

To this end, the view frustum will be divided into a coarse three-dimensional grid and each view sample will be processed to mark those grid-cells that are occupied. From this grid we then build a hierarchy against which we can traverse triangle shadow-volumes. A shadow volume can be tested against any node in the hierarchy as the bounds of the corresponding AABB are implicitly defined. When a leaf node is found to be intersecting with a shadow volume, all view samples that reside in the same two-dimensional tile as that node are tested. We show that performance can be further improved by calculating the explicit bounds of each node, and that these can be efficiently calculated while building the hierarchy. The

main improvement over previous work comes from this much tighter acceleration structure, with which a shadow volume will only need to traverse down to the leaf nodes if it actually lies very close to the samples contained therein.

We will begin by describing the basic steps of our algorithm, starting with an overview and then discussing the different parts in detail. We will then discuss some shortcomings of our initial implementation and how they can be overcome. Our algorithm and implementation closely follow the steps of the PTSV algorithm detailed by Sintorn et al. [2011]. We have implemented the algorithm in CUDA, and it runs entirely on the GPU, without reading any data back to the CPU. The steps of the algorithm are:

- **Building a Hierarchy** Using the current G-Buffer, build an acceleration structure that groups view samples that are close to each other.
- **Triangle Setup** For each shadow casting triangle, create its shadow volume, i.e., a set of planes that enclose the volume of space that is in shadow due to that triangle.
- **Traversal** Traverse each triangle shadow volume through the hierarchy, culling nodes that lie completely outside and marking those that are completely inside the volume as *in shadow*. Only when a node might intersect the shadow-volume planes do we traverse into its children.

Building the acceleration structure As in the PTSV algorithm, we choose to build a tree that has a branching factor of 32. This allows the SIMD lanes of one multiprocessor of the GPU to work in parallel with the intersection tests that make up the bulk of our traversal algorithm, and so we can utilize the hardware efficiently. Using another fanout is a trivial change to the algorithm, however, so it could easily be varied for different hardware.

We chose to group view samples into clusters that are 8×8 pixels. Thus, a single cluster can contain a maximum of 64 view samples. Given a view sample whose pixel coordinates are (x, y) and which has a view space depth z, we calculate the *cluster coordinate*, $\mathbf{x}'$, as:

$$\mathbf{x}' = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \lfloor x/8 \rfloor \\ \lfloor y/8 \rfloor \\ \left\lfloor \frac{log(-z/near)}{log(1+\frac{2tan\Theta}{S_y})} \right\rfloor \end{pmatrix}, \quad (1)$$

where $near$ is the distance to the near plane, Θ is the field-of-view, and S_y is the number of cluster divisions in height. The cluster coordinate's z' component is chosen as in the work by Olsson et al. [2012], i.e., we subdivide the frustum exponentially in depth to obtain leaf nodes whose implicit bounds are frustums with a depth that is roughly equal to their width and height in world space.

We interleave these integer coordinates to produce a key in morton order [Morton 1966], which we call the *cluster key*. This key uniquely defines a leaf node in our hierarchy, and several view samples may have the same key. For now, we will consider a screen resolution of 1024×1024, and so we only need seven bits for the x' and y' coordinates. The number of bits needed for the z' component depends on the ratio between the near and far distances used in the projection. At this resolution, we found nine bits to be more than sufficient for all of our scenes. We therefore rearrange the cluster key somewhat, as shown in Figure 2, to allow for a more shallow tree as discussed below.

We now need to build an acceleration structure over these keys. In our initial attempts, we followed the approach of Olsson et al. [2012], building a list of cluster keys and compacting this list so that we had a

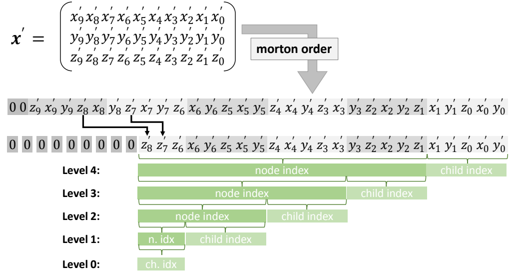


Figure 2: *The cluster coordinate is packed into a 23-bit integer using a slightly rearranged morton order. This key can then efficiently be used to populate a full tree hierarchy.*

minimal set of clusters from which to build the tree. However, when building a hierarchy on top of these clusters, we need to maintain a pointer from each node to where its children are stored, along with a bitmask that tells us which of the children exist. While we were able to build this compact list and hierarchy very quickly ($<$ 1ms for a resolution of 1024×1024), both building the tree and traversing it is significantly faster if we choose to sacrifice some memory and store a full tree instead.

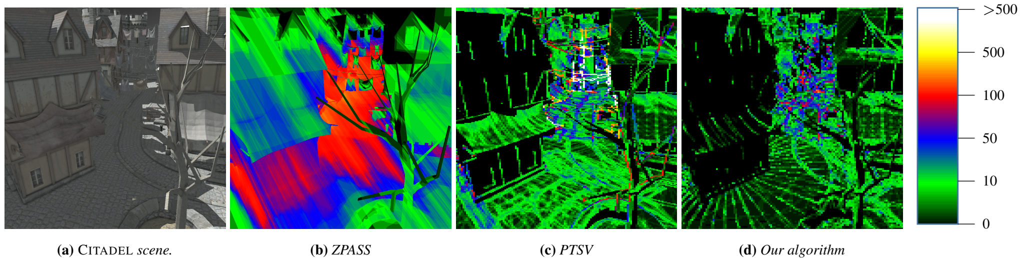
The five levels of our hierarchy are represented by five arrays of 32-bit words. Each word is a child mask, where a set bit indicates that the corresponding child exists in the tree. To build the final level of the tree (where a set bit indicates the existence of a leaf node, or cluster), we can simply process all view samples and calculate their cluster keys. The 18 most significant bits of the key give us the index to the node in the level-four array in which the view sample resides. The five least significant bits tell us which of the bits of this node shall be set to indicate that this cluster exists. Having populated the lowest level of our tree, we shift the cluster key 5 bits to the right and repeat the process to populate the next level above. This goes on until we have a single root node at level 0 represented by a single 32 bit word (see Figure 2).

Our initial implementation of this step is to simply start a thread per view sample and let that thread atomically update all the nodes in which it resides (in Section 2.1, we will add optimizations).

Triangle Setup For each triangle, we calculate the four planes that enclose a triangle shadow volume. Care must be taken to ensure that two triangles that share an edge will calculate exactly the same plane for that edge, and when not using light front-face culling, a small bias must be added to the triangle plane to avoid self shadowing. We transform these planes into clip space, using the camera's model-view-projection matrix, and store them in a list. This basic triangle setup is exactly the same as is done in PTSV.

These four planes are sufficient to produce correct results, but will, during traversal, produce a lot of false positives. In PTSV, tiles are also culled against the two 2D silhouette edges of the shadow volume, which alleviates the problem. We have found that this method still produces a significant amount of false positives (see Figure 6) and suggest another set of culling planes.

The problem with false positives occur when an Axis Aligned Bounding Box (AABB) does not lie completely outside all shadow volume planes. This can happen in the wedges formed by any two of the planes (see Figure 4). For each vertex (and so, each wedge), we add a plane that contains both the light source and the vertex. Any rotation of that plane is legal as long as it does not intersect the



(a) CITADEL *scene.* **(b)** *ZPASS* **(c)** *PTSV* **(d)** *Our algorithm*

Figure 3: *Heat maps showing the number of times each pixel is tested against a shadow volume. The PTSV algorithm has a very unbalanced workload around steep depth discontinuities, where the view-sample cluster hierarchy performs much better.*

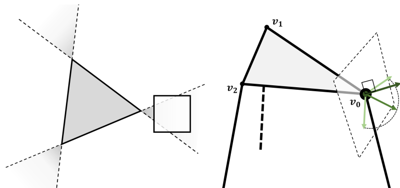


Figure 4: *Left: False positives occur when a node that does not intersect the triangle still does not lie completely outside any plane. Right: We add new planes, attempting to cull nodes that fall within the false-positive wedge.*

shadow volume. We choose to create the half vector between the two shadow-volume plane normals, make that orthonormal to the vector pointing from the vertex to the light source, and use that as the plane normal for our culling plane.

A node that lies completely outside any of these new planes is guaranteed to lie outside the shadow volume, and testing against all three planes improves culling (and performance) significantly. Note that for degenerated triangles or triangles whose normals are close to orthogonal to the light direction, these calculations can be unstable, and so we simply skip these culling planes in such cases.

Traversal The traversal kernel is written in a *persistent-threads* fashion, where we start enough warps to keep the machine fully utilized and let each warp pick jobs from a list by atomically updating a counter to get an index. Each job, in this case, is one triangle shadow volume, which all threads in the warp will cooperatively traverse against the hierarchy, starting with the root node. Each thread will consider one child of the current node and test that child against the four planes that make up the shadow volume. The thread will first consult the child mask for the current node to make sure that the child exists in the tree. If it does, but the child is completely outside any of the shadow volume planes, it is rejected. If the child lies completely inside all four planes, it can be trivially accepted, which is noted by simply clearing the corresponding bit in the current nodes child mask. If neither is true, the child is tested against the extra culling planes described above and if it does not lie completely outside any of those, it is considered intersecting.

After the intersection test, the results are broadcast (using CUDAs ballot instruction) so that all threads know which children are intersected. These will now be tested in turn in the same way. In order to know where in the tree we are, we maintain a current *key* throughout the process (as detailed in Algorithm 1).

When the leaf-node level is reached, the threads will instead cooperate in testing the individual view samples within the tile. At this level, we have no information about which view samples actually lie in the cluster and which lie in another cluster with the same 2D bounds, so we simply test all samples and update the shadow buffer if the sample lies within the shadow volume. We could of course store this extra information while building the hierarchy, but as 32 intersection tests will always be done in parallel, it seems unlikely that this would improve overall performance.

So far, our traversal algorithm does not differ much from that of Sintorn et al. [2011]. One difference is that their acceleration structure *is* a full tree with samples in every node, whereas, while we store our structure as a full tree, it is actually very sparse, and so we need to fetch the child mask for each node in order to know which children to test. Another difference lies in how the clip-space coordinates for the AABBs of the nodes are calculated. Unlike their algorithm, in which the z-components are fetched from a texture, we get all coordinates implicitly from the cluster keys. The traversal algorithm is outlined in Algorithm 1. We have implemented this algorithm in CUDA, both in an iterative fashion, using a small stack, and as written in Algorithm 1, using template metaprogramming for the recursion. The latter performs slightly better, probably due to better optimization opportunities.

After traversal, we will know for each pixel whether it is in shadow or not, *except* for pixels that have only been trivially accepted as part of some node. We must therefore run one final pass where we start one thread per view sample. Each thread will perform almost exactly the same job as when building the hierarchy (as explained above), except that this time, instead of updating the hierarchy, it will just make sure that all the nodes it resides in are still marked as existing. Otherwise, the node has been trivially accepted and the view sample is set as shadowed.

Remaining problems The algorithm, as described so far, works well and we can see from our measurements that we have mostly eliminated the problem where the amount of work increases significantly at steep depth discontinuities (see Figure 6). Unfortunately, we can also see that the total number of test and set operations that we need to do each frame is overall significantly higher than in PTSV. This is partly due to us having a deeper tree but also because the size of our nodes is completely unaffected by the actual view samples within. In PTSV, tiles that do not contain depth-discontinuities will have a fairly well fitting bounding box, while ours will be fixed size and potentially very conservative.

Another problem, which we share with PTSV but which will be more exaggerated in our case, is the possibility of poor load balancing. Even with our persistent threads model, we see that some

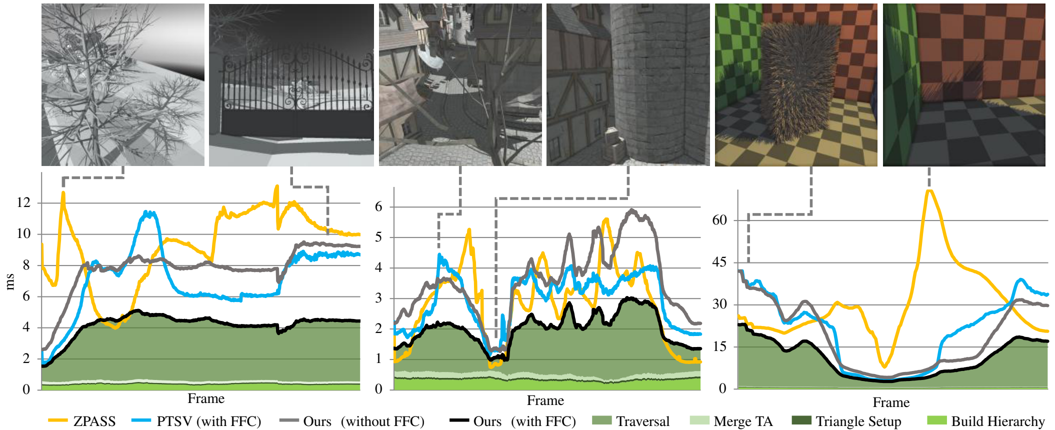


Figure 5: *Time taken by each part of our algorithm, along with total time taken for ZPASS and PTSV, for three different animations. Total performance of our algorithm is presented with and without Front Face Culling (FFC). From left to right,* VILLA *(88k triangles),* CITADEL *(60k triangles) and* FUZZY *(400k triangles).*

multiprocessors have to wait for a long time, while a few finish processing triangles that generate much more work than the average. This problem is illustrated in Figure 7.

Finally, where PTSV only has to build a small mip-map hierarchy over the depth buffer, we have to build a much larger hierarchy. Initially, we did this as explained above, with one thread per view sample, which proved to be a less than optimal solution. We will discuss these three problems in the upcoming sections.

2.1 Explicit bounds

In our hierarchy, the existence of a node is indicated by a single bit in its parent's child mask. The size of each node is implicitly defined by its position in the tree. This makes for a very memory-efficient representation, but, as explained above, does not give us much opportunity to cull shadow volumes. We will now explore the possibility of storing a complete AABB with each node to improve culling performance. Building the hierarchy with AABB information is not much more complicated than without (although doing so efficiently takes some extra thought, as detailed below). We can simply let each view sample atomically update all of its parents bounding boxes. As we currently store a full tree, the memory requirements are however significantly increased (see Section 3 for details).

With explicit bounds, the performance of the traversal is dramatically improved (see Figure 8). This is partly due to the traversal kernel becoming much simpler, as we no longer have to calculate the implicit trivial-reject and trivial-accept points for each node, but much more due to the significant decrease in volume of our acceleration structure (see Figure 1c). Previously, a node which only contained a few samples could still be very large, and a shadow-volume that touched that node would have to traverse all of its children. Now, each node will have a bounding box which tightly fits the contained view samples. The number of test-and-set operations required are now almost always fewer than in PTSV, despite our deeper tree. In frames with steep depth discontinuities, we often perform better than a factor of $2\times$ fewer operations (see Figure 6).

2.2 Load Balancing

As illustrated in Figure 7, a small fraction of the triangles may require much more work than the average, which leads to very poor load balancing with our method. To combat this, we simply split the traversal step into two parts. In a first pass, each warp will pick one triangle as before and will then traverse down to a predefined level, L. Every time the traversal reaches that level, it will atomically push a new job onto a list and then return to traverse the rest of the upper part of the tree. This job is simply a tuple (t, k) where t is the triangle ID and k is the key of the node in which traversal was aborted.

In a second pass, each warp will instead pick a job from the newly created list and start traversal where the previous pass left off. Note that we do not have to retrace the path taken to reach the node, but can immediately continue traversal. Therefore, the only additional work that is required by this two-pass method is writing and reading the intermediate job list which fortunately turns out to be quite modest in size even if we let the first pass go all the way down to the leaf-node level. The performance gain from improved load balancing is sometimes very large (as can be seen in Figure 8), and the two-pass method has a much more stable performance.

2.3 Efficiently building the hierarchy

Building the hierarchy in the way described so far (by launching one thread per view sample) leads to very high contention, as many threads will attempt to update the same node. We also perform a lot of redundant work, as several threads will consider view samples that lie in the same cluster and will all mark that cluster as existing. In fact, after having implemented the optimizations described above, building the hierarchy is actually often the most time-consuming part of the algorithm. We reduce the amount of redundant work by launching one warp per 8×8 pixel tile. While all view samples in this tile *could* lie in different clusters, they will usually occupy only a few. The threads within the warp will cooperate in finding the bounding box of each occupied cluster, and then, a single thread can atomically update the nodes in global memory. To reduce the

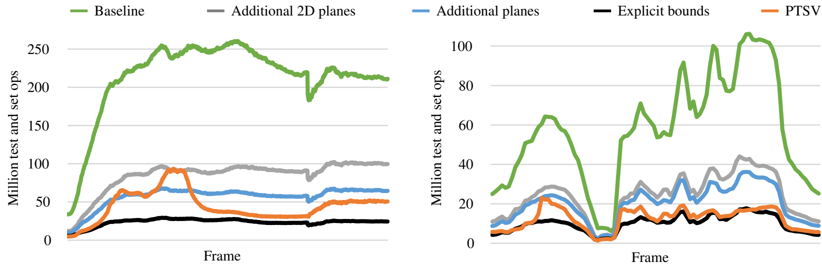


Figure 6: *Number of test-and-set operations required by various versions of our algorithm and by PTSV, for animations in two different scenes (left is* VILLA *and right is* CITADEL*). Culling of shadow-casting triangles that face the light source is enabled in these experiments.*

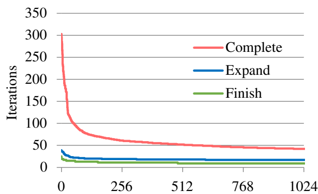


Figure 7: *Number of traversal iterations performed in the 1024 largest jobs. The red line shows the original algorithm where each triangle is processed by a single warp. When we turn to a two-pass algorithm, no single warp will have to do many more iterations than the average.*

amount of contention in the top of the tree, we perform the hierarchy construction in two passes. A first pass will build the tree up to an intermediate level (as described above), and in a second pass, we launch a warp per node at this intermediate level to perform the atomic insertion of bits, and updating of AABBs, upwards to the root. As can be seen in Figure 5, building the hierarchy is now extremely fast. In fact, it takes about the same time as building the hierarchical depth buffer in PTSV.

3 Discussion and Results

We have measured the performance of our algorithm using animated fly-throughs of three different scenes. CITADEL is the scene used to measure performance in the PTSV paper, with 60k triangles. VILLA is a scene designed to stress shadow volume algorithms, with 88k triangles. Finally, FUZZY is a more complex scene with 400k very dense shadow-casting triangles. All experiments were done on an NVIDIA Titan GPU with a screen resolution of 1024×1024 unless otherwise stated. In Figure 5, we show the total time taken by each part of our algorithm, and we can see that we pay a fairly low and constant price for building the hierarchy (Build Hierarchy), calculating the shadow volume planes (Triangle Setup) and finding view samples that have been trivially accepted as part of some node (Merge TA). The cost for traversing the shadow volumes against the hierarchy is naturally view dependent, but the worst-case performance seems to be much more stable than in previous work. In the same plot, we see the performance of the PTSV algorithm and our carefully tuned ZPASS implementation. We can see that our algorithm outperforms both, except in views where there are very few shadow volumes on screen, where ZPASS performs better. It is interesting to note that, unlike the measurements provided in the PTSV paper (which were done on a GTX480 GPU), PTSV seems to perform as well as ZPASS throughout the CITADEL sequence. This is probably due to ZPASS being entirely limited by the pixel throughput of the GPU, which has not changed much between the GTX480 and the Titan GPUs. We measured the performance of our ZPASS implementation on a GTX480 as well and found that performance was indeed very similar.

As in PTSV, we can use *Front Face Culling (FFC)* of shadow-casting triangles when objects are closed, which improves performance significantly. In Figure 5, we also present the total time taken by our algorithm when this optimization is turned off. All timings reported for PTSV are as obtained with the optimization applied.

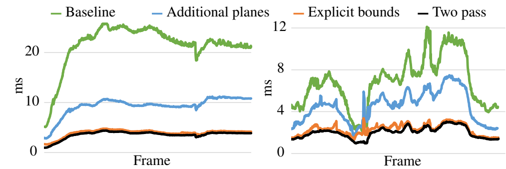


Figure 8: *Time taken by the traversal part of our algorithm when applying different optimizations (left is* VILLA *and right is* CITADEL*).*

In Figure 8, we show the time taken by the traversal stage of our algorithm with different optimizations applied. The Baseline is where we only test implicit bounding boxes against the basic shadow-volume planes. We then add the additional culling planes, as described in Section 2. We improve performance significantly again by calculating the explicit bounding boxes for each node, and finally remove spikes that are due to poor load balancing by performing the traversal in two passes.

The problem with load balancing is illustrated in Figure 7. The lines here show the number of iterations performed in the traversal algorithm for the 1024 most demanding jobs when rendering the view shown in Figure 1. For the original algorithm (Complete), we can see that there are a few triangles that have much more work to do ($\sim 10\times$ average), and therefore, most multiprocessors will have to idle until one multiprocessor has finished the last of these. When we instead divide the work into two passes (Expand and Finish), the number of iterations are more evenly distributed and we avoid sudden unexpected spikes in rendering time.

We have measured the number of test-and-set operations required by our algorithm, as it provides an implementation-independent metric of the efficiency of the algorithm. In Figure 6, we have plotted this metric for our algorithm (with different optimizations applied) and for the PTSV algorithm. As expected, we see that our algorithm is much less sensitive to high-frequency depth buffers and that it performs an equal amount of work as, or less work than, the PTSV algorithm throughout the animations. While the improvement from our additional culling planes is modest in the CITADEL scene, it helps significantly in the VILLA scene, which contains many more small shadow casting triangles.

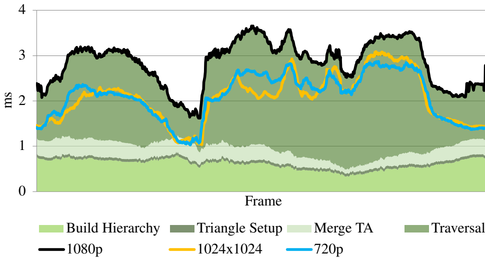


Figure 9: *Time taken by various parts of our algorithm when rendering the* CITADEL *animation to resolution 1080p and the total time when rendering to resolutions 720p and 1024×1024.*

When rendering to higher resolutions, we must make a few small changes to the algorithm. First, our implementation as described so far has fairly high memory requirements due to storing bounding boxes for the hierarchy as a full tree. With a maximum resolution of 1024×1024, a cluster size of 8×8, and with the maximum number of z' bits in the cluster key set to 9, the total number of nodes in a full tree is 8.65 million. The hierarchy information, with a single bit per node, costs only about 1 MB of memory. But if we store six floats per bounding-box, the cost is 198MB, which might be too much in some cases. We have implemented an alternative version, where we only store a pointer per node (33MB) and a compact list of only the existing bounding boxes. The compact list never exceeds 52000 elements in our tests (but the worst case size is 1 million elements), and this version runs almost as fast as the original (at most 5% slower). With this alternative version, then, the expected working memory requirements are around $(33 + 1)$MB and worst case is $(33 + 25)$MB. We can now support a full HD resolution with expected $(66 + 2)$MB and worst case $(66 + 50)$MB.

Secondly, to support resolutions larger than 1024 pixels wide or high, we must store a longer morton key than the one described in Figure 2. When rendering to 1080p, we simply add one bit each for the x, y and z components, which increases the largest supported resolution to 2048×2048, with the same depth range. With a 32-bit morton code we could support resolutions up to 8192×8192.

In Figure 9 we show the same timings as in Figure 5, but when rendering to a resolution of 1920×1080. For reference, we have also plotted the total time taken when rendering to resolutions 1280×720 and 1024×1024. All timings use the more compact memory layout for bounding boxes described above. As expected, building the hierarchy takes approximately twice as long as for the lower resolutions, since it will contain roughly twice the number of nodes. Traversal scales much better however, since many more nodes can be trivially accepted. The slight differences in running time when comparing the two lower resolutions are mostly due to the images having different horizontal field-of-views.

Acknowledgements

This work was supported by the Swedish Foundation for Strategic Research under Grant RIT10-0033. The TITAN GPU used for this research was donated by the NVIDIA Corporation. The CITADEL scene is a part of the Epic Citadel level distributed with the Unreal Development Kit by Epic Games.

References

AILA, T., AND AKENINE-MÖLLER, T. 2004. A hierarchical shadow volume algorithm. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, HWWS '04, 15–23.

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proc. of EGSR 2004*, 161–166.

BILODEAU, W., AND SONGY, M., 1999. Real time shadows. Creativity 1999, Creative Labs Inc. Sponsored game developer conferences, Los Angeles, California, and Surrey, England.

CARMACK, J., 2000. Z-fail shadow volumes. Internet Forum.

CHAN, E., AND DURAND, F. 2004. An efficient hybrid shadow rendering algorithm. In *Proc. of the EGSR*, 185–195.

CROW, F. C. 1977. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11* (July), 242–248.

EISEMANN, E., SCHWARZ, M., ASSARSSON, U., AND WIMMER, M. 2011. *Real-Time Shadows*. A.K. Peters.

HARADA, T. 2012. A 2.5D culling for forward+. In *SIGGRAPH Asia 2012 Technical Briefs*, ACM, New York, NY, USA, SA '12, 18:1–18:4.

HEIDMANN, T. 1991. Real shadows, real time. *Iris Universe 18*, 28–31. Silicon Graphics, Inc.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. on Graphics 24*, 4, 1462–1482.

LAINE, S. 2005. Split-plane shadow volumes. In *Proceedings of Graphics Hardware 2005*, Eurographics Association, 23–32.

LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution matched shadow maps. *ACM TOG 26*, 4, 20:1–20:17.

LLOYD, B., WEND, J., GOVINDARAJU, N. K., AND MANOCHA, D. 2004. Cc shadow volumes. In *EGSR/Eurographics Workshop on Rendering Techniques*, 197–206.

MORTON. 1966. A computer oriented geodetic data base and a new technique in file sequencing. Tech. Rep. Ottawa, Ontario, Canada.

OLSSON, O., AND ASSARSSON, U. 2011. Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4, 235–251.

OLSSON, O., BILLETER, M., AND ASSARSSON, U. 2012. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGGH-HPG'12, 87–96.

SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample-based visibility for soft shadows using alias-free shadow maps. *CG Forum (EGSR 2008) 27*, 4 (June), 1285–1292.

SINTORN, E., OLSSON, O., AND ASSARSSON, U. 2011. An efficient alias-free shadow algorithm for opaque and transparent

objects using per-triangle shadow volumes. *ACM Trans. Graph. 30*, 6 (Dec.), 153:1–153:10.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12* (August), 270–274.

ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *Proc. of the 2006 ACM international conf. on Virtual reality continuum and its applications*, VRCIA '06, 311–318.

A Appendix

Here, we provide pseudocode that describes the traversal algorithm suggested in this paper. This code is run cooperatively by all threads in a warp. The TRAVERSENODE procedure is started for each triangle shadow volume (SV), with *level* and *key* initially set to 0, and *childMask* set to the root node childMask. MAXLEVELS is the total number of levels of the tree (6 with our layout for 1024×1024).

Algorithm 1 Pseudocode describing how the view-sample cluster hierarchy is traversed for a triangle's shadow volume. The symbols $\gg$ and $\ll$ denote right and left shift. & denotes bitwise AND. ! denotes bitwise invert.

```
 1: procedure TRAVERSENODE(SV, level, key, childMask)
 2:     if level = MAXLEVELS-1 then
 3:         TESTVIEWSAMPLES(SV, key)
 4:         return
 5:     nodeBitPos ← (MAXLEVELS − level + 1) * 5
 6:     childBitPos ← (MAXLEVELS − level) * 5
 7:     childKey ← key|(laneId ≪ childBitPos)
 8:     intersect ← true
 9:     trivialAccept ← true
10:     if TESTBIT(laneId, childMask) then
11:         for each plane in SV do
12:             p ← CLIPSPACETRPOINT(childKey, plane)
13:             if p dot plane > 0 then
14:                 intersect ← false
15:                 trivialAccept ← false
16:                 break
17:             p ← CLIPSPACETAPOINT(childKey, plane)
18:             if p dot plane < 0 then
19:                 trivialAccept ← false
20:     intersectionResult ← BALLOT(intersect)
21:     TAResult ← BALLOT(trivialAccept)
22:     if laneID = 0 and TAResult ≠ 0 then
23:         nodeIdx ← key ≫ nodeBitPos
24:         ATOMICAND(hierarchy [level] [offset] , !TAResult)
25:     childMask ← intersectionResult&!TAResult&childMask
26:     while childMask ≠ 0 do
27:         nextChild ←31-CLZ(childMask)
28:         nextKey ← key|(nextChild ≪ childBitPos)
29:         nextNodeIdx ← nextKey ≫ childBitPos
30:         nextChildMask ← hierarchy [level+1] [nextNodeIdx]
31:         TRAVERSENODE(SV, level+1, nextKey, nextChildMask)
32:         UNSETBIT(nextChild, childMask)
```

TESTVIEWSAMPLES is run when the final level is reached and simply tests the individual view samples of a tile in parallel. CLIPSPACETRPOINT and CLIPSPACETAPOINT find the trivial-reject and trivial-accept points respectively of a node's AABB. When using implicit bounds, this is done by calculating the AABB from the provided key, and with explicit bounds, the key is used to look up a bounding box in memory. TESTBIT(a, b) returns true if bit a is set in word b. UNSETBIT(a, b) clears bit a in word b. CLZ(a) is the CUDA intrinsic that counts leading zeroes in word a. BALLOT(a) is the CUDA warp vote function, which returns a word with bit b is set if a is *true* for thread b.