Fast and High-Quality Visibility Determination

Rasmus Barringer
Department of Computer Science
Lund University

LUND INSTITUTE OF TECHNOLOGY
Lund University

ISBN 978-91-7623-375-7 (Printed)
ISBN 978-91-7623-376-4 (Electronic)
ISSN 1404-1219
Dissertation 47, 2015
LU-CS-DISS:2015-03

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: rasmus.barringer@cs.lth.se
WWW: http://www.cs.lth.se/Rasmus_Barringer

Typeset using LaTeX2ε
Printed in Sweden by Tryckeriet i E-huset, Lund, 2015

Abstract

Computer generated imagery is vital to the entertainment industry in the production of games and films, for example. It is also increasingly important for visualization in design, architecture, engineering, and medicine, to name a few. Improvements to rendering techniques come from a combination of improved algorithms and more powerful hardware. Typically, hardware developments introduce new challenges and opportunities for algorithms that better fit the platform. Recently, these developments involve a widening gap between memory bandwidth and compute capabilities, wide SIMD units, and shared memory between CPU and GPU.

The focus of this thesis is on improved algorithms for visibility queries that are used in graphics, motivated by challenges introduced by recent hardware developments. Geometry sampling lies at the core of rendering techniques, both for real-time and offline rendering, and generating images of higher quality generally involves taking more samples. Performance and quality improvements for visibility samples can thus enable higher quality rendering within a smaller time budget.

This thesis presents five published papers with new solutions for visibility queries in the two major rendering paradigms in use today: rasterization and ray tracing. Two-dimensional rasterization is common in real-time graphics because of its computational efficiency. Multiple point samples are usually taken for each pixel to get high-quality images without aliasing artifacts. When rendering thin curves, many samples will typically be required to achieve acceptable quality. In this context, we propose that thin curves can be rasterized in high quality on a graphics processor using spatial line samples and curve-specific intersection tests. Further, we propose that the recent advent of shared memory between CPU and GPU can allow for MSAA computations to be offloaded to idle CPU cores. In three-dimensional rasterization, specifically rasterization with motion blur, we introduce a way to render practically noise- and alias-free images with competitive performance using semi-analytical line-based visibility queries on a multi-core CPU.

Ray tracing, common in offline rendering, is a flexible rendering technique that can model how light propagates in a scene. Challenges in ray tracing include how a ray can be efficiently tested against a scene. Tests are accelerated by building a spatial hierarchy over the scene and our work in ray tracing specifically targets the process of traversing rays against a bounding volume hierarchy (BVH). The first contribution involves a flexible BVH traversal algorithm that executes without storing the traversal state in a stack, which may be beneficial in cases where storing a stack is expensive. The second contribution is an efficient algorithm for traversing large streams of rays against a BVH while making use of wide SIMD.

Acknowledgements

I would like to acknowledge my main supervisor, Tomas Akenine-Möller, who is a big reason for why I got interested in graphics research. His ideas, intuitions, hard work, and trust have made my research possible and have allowed me to grow as a person. I give thanks to Carl Johan Gribel for our collaborative efforts in analytical rendering and for helping me get a good start in research. My assisting supervisor, Michael Doggett, has my gratitude for solid advice and for our joint projects. I would also like to thank the other members of the graphics group at Lund University for interesting and valuable discussions over numerous and tasty Grafikas: Per Ganestam, Magnus Andersson, and Björn Johnsson. The people who helped make my two internships at Intel fantastic experiences also receive my gratitude: Tomas Akenine-Möller, Petrik Clarberg, Jacob Munkberg, Jon Hasselgren, Robert Toth, Charles Lingle, Marco Salvi, Karthik Vaidyanathan, and Gabor Liktor. Acknowledgement goes to my family and friends for their support and for showing genuine interest during my graphics-related ramblings. Finally, I would like to acknowledge my wife My, who have kept inspiring and encouraging me through long crunches with early-morning deadlines.

Preface

This thesis summarizes my research in visibility computations for graphics. The following papers are included:

I. Carl Johan Gribel, Rasmus Barringer, and Tomas Akenine-Möller, "High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility", in *ACM Transactions on Graphics*, 30(4):54:1–54:12, 2011.

II. Rasmus Barringer, Carl Johan Gribel, and Tomas Akenine-Möller, "High-Quality Curve Rendering using Line Sampled Visibility", in *ACM Transactions on Graphics*, 31(6):162:1–162:10, 2012.

III. Rasmus Barringer and Tomas Akenine-Möller, "Dynamic Stackless Binary Tree Traversal", in *Journal of Computer Graphics Techniques*, 2(1):38–49, 2013.

IV. Rasmus Barringer and Tomas Akenine-Möller, "A^4: Asynchronous Adaptive Anti-Aliasing using Shared Memory", in *ACM Transactions on Graphics*, 32(4):100:1–100:10, 2013.

V. Rasmus Barringer and Tomas Akenine-Möller, "Dynamic Ray Stream Traversal", in *ACM Transactions on Graphics*, 33(4):151:1–151:9, 2014.

Contents

1 Introduction 1
2 Measuring Image Errors 4
3 Hardware Platforms 5
3.1 CPU 5
3.2 GPU 7
3.3 Shared Memory 8
4 Contributions and Methodology 8
5 Visibility Determination 9
5.1 Two-Dimensional Rasterization 10
5.2 Higher-Dimensional Rasterization 17
5.3 Ray Tracing 19
6 Conclusions and Future Work 22
Bibliography 23

Paper I: High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility **27**
1 Introduction 29
2 Previous Work 30
3 Algorithm Overview 31
4 Depth Patches 34
5 Visibility Engine 35
5.1 Binning 36
5.2 Depth Sorting 36
5.3 Pixel Integration 36
6 Ambient Occlusion 37
6.1 Static Ambient Occlusion 38
6.2 Motion Blurred Ambient Occlusion 40

7 Implementation 41
8 Results 42
9 Conclusions and Future Work 46
Bibliography 51
A Depth Patch Approximation 54
A.1 Adaptive Refinement of the Approximation 56
B Shading Cache 57

Paper II: High-Quality Curve Rendering using Line Sampled Visibility 59
1 Introduction 61
2 Previous Work 62
3 Algorithm Overview 63
4 Visibility Engine 64
4.1 Thin Curve Representation 65
4.2 Thin Curve/Line Sample Intersection 66
4.3 Visibility for Large Projected Curve Widths 69
4.4 Interval Resolve Procedure 71
5 GPU Implementation 72
5.1 Setup 73
5.2 Binning 74
5.3 Rasterization 75
5.4 Sample Blend 75
6 Results 76
7 Conclusions and Future Work 81
Bibliography 82
A Compact Representation 86

Paper III: Dynamic Stackless Binary Tree Traversal 87
1 Introduction 89
2 Implicit Traversal Algorithm 90
2.1 Left-first Traversal 91
2.2 Generalized Traversal 92
3 Sparse Traversal Algorithm 95
4 Results 95
5 Conclusion and Future Work 97
Bibliography 100

Paper IV: A^4: Asynchronous Adaptive Anti-Aliasing using Shared Memory **103**

1 Introduction . . . 105
2 Previous Work . . . 105
3 Algorithm Overview . . . 107
4 GPU Rendering Pipeline . . . 109
5 CPU Rendering Pipeline . . . 109
5.1 Silhouette Edge Detection . . . 110
5.2 Sparse Rasterization . . . 111
5.3 Sparse Anti-Aliasing . . . 112
6 Implementation . . . 115
7 Results . . . 117
8 Conclusions and Future Work . . . 123
Bibliography . . . 125

Paper V: Dynamic Ray Stream Traversal **127**

1 Introduction . . . 129
2 Previous Work . . . 130
3 Overview . . . 131
4 Traversal Algorithm . . . 133
4.1 BVH2 . . . 134
4.2 BVH4 . . . 137
5 Implementation . . . 138
6 Results . . . 140
7 Conclusions and Future Work . . . 145
Bibliography . . . 146
A Core Loop Implementation for BVH2 . . . 150

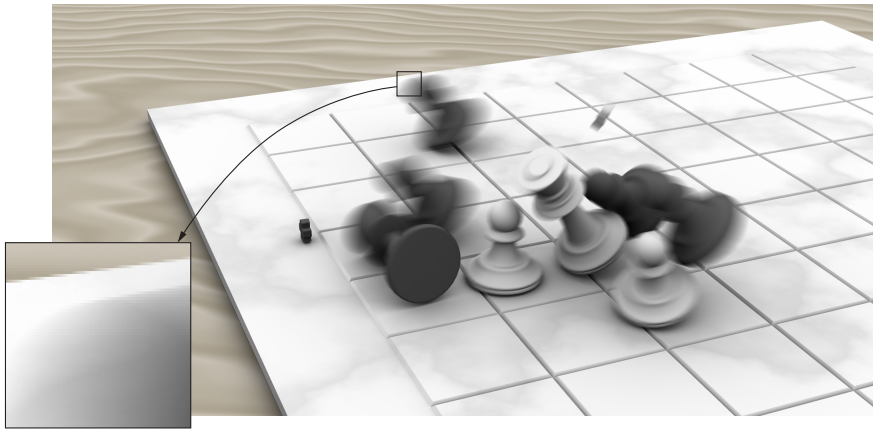


Figure 1: *Moving chess pieces with motion blur, procedural textures, and a simple occlusion-based light model.*

1 Introduction

The focus of this thesis is on advancements of visibility queries, which are used extensively in computer graphics, and more specifically in computer generated imagery (CGI). Computer generated imagery is the process in which computer graphics is used to create images, printed media, video games, films, and television programs, to name a few. An example of such an image is shown in Figure 1.

In order to generate an image, a model of some virtual space to be converted to imagery is needed. The process of converting the model to a final image is usually denoted *rendering*. In this work, the focus is on rendering virtual spaces with three spatial dimensions, with the possible addition of time as an extra dimension (see Section 5.2 for details). The final image is typically made up of a rectangular grid of individual *pixels*, where each pixel contains a color value for its portion of the grid. This representation is known as raster graphics. In the case of rendering videos or interactive applications, a single image is referred to as a *frame*.

In a three-dimensional space, a model is needed for virtual objects, their appearance in the scene (i.e., how light interacts with their surfaces) as well as a camera model that describes how light is received by the virtual camera sensor. A simple example is shown in Figure 2. A common representation for an object is a collection of triangles, which is often denoted a *mesh*. Triangles are common since it is the simplest polygon in three dimensions that make up a surface, with the added benefit that this surface always is planar. The mesh representing one of the chess pieces is shown in Figure 3.

How the appearance of an object is modelled depends highly on how light is simulated in the scene. In *real-time rendering* [3], which is used in games for example, performance is of outmost importance with a typical time budget of 16 millisec-

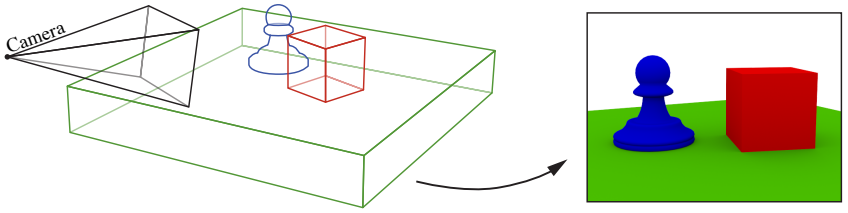


Figure 2: *A model of a three-dimensional scene consisting of three objects and a camera. The result of rendering the scene from the camera's point of view is shown to the right.*

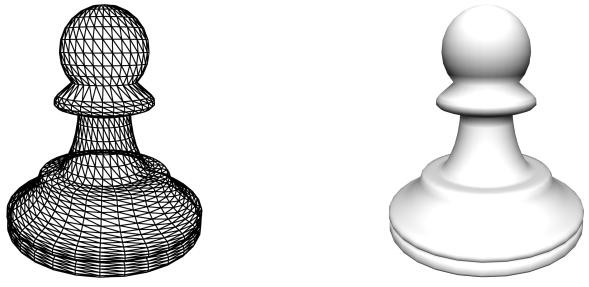


Figure 3: *Left: the triangular mesh representation of a chess piece. Back facing triangles are hidden. Right: the same mesh after rendering.*

onds per frame. Here, it is common to project triangles to the camera's point of view and determine what is directly in front of each pixel in the image, a process known as *rasterization*. Projection is performed by transforming the *vertices* that define the triangles (three points per triangle) using a 4×4 matrix. The rasterization stage then determines overlap between pixels of the image and the triangles. The output is one *fragment* for each triangle-pixel pair that overlaps. Each fragment contains the information necessary to compute the color for that specific part of the triangle. On modern graphics hardware, determining the color of a fragment amounts to running a specialized *shader* program [29], thus allowing for a great deal of flexibility in how materials are modelled. When multiple triangles overlap the same pixel, the rendering pipeline needs to determine which fragment is closest to the camera. In conventional two-dimensional rasterization, this visibility problem is usually solved using a z-buffer [7, 34]. As an optimization, testing against the z-buffer can often be performed before shading, thus reducing the number of fragments to shade. While programmable shading allows for a wide range of materials to be modelled, they are ultimately limited by the strict timing requirements imposed by real-time rendering. *Offline rendering* can, on the other hand, spend hours rendering a single frame, and may thus afford more involved models

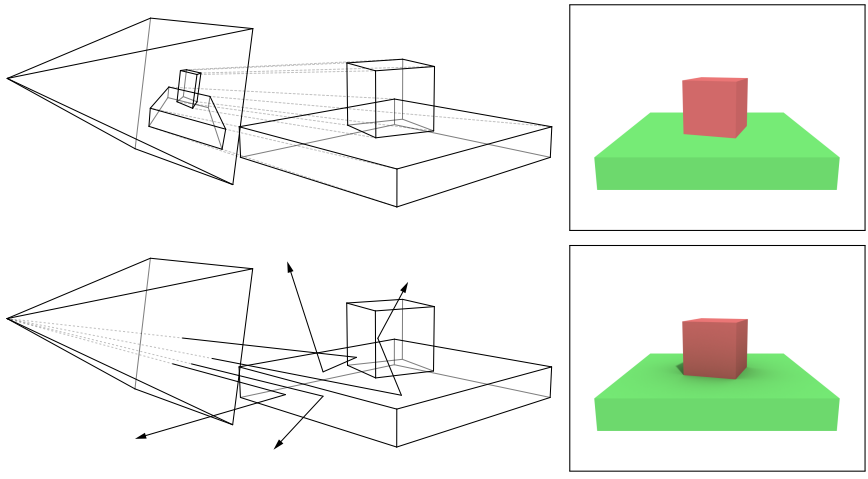


Figure 4: *Top: rasterization - geometry is projected on the image plane. Polygon to pixel overlap can then be determined using simpler tests in two dimensions. This is a very efficient method for determining what is immediately in front of each image pixel. Bottom: ray tracing - rays are sent from the camera through pixels in the image plane. Each ray determines overlap with the scene in three-dimensional space. This is often a more expensive, but also more flexible, way of determining visibility. When tracing rays, it is possible to continue the path of the ray in the scene randomly in order to simulate how light interacts between objects. This easily captures effects such as diffuse interreflections and soft shadows, for example.*

for light transport within a scene. While forms of rasterization still are important in this context, they typically involve higher-dimensional stochastic rasterization, described in Section 5.2. The trend seems now to be a push for more *ray tracing* based solutions in offline rendering since it allows for more realistic, albeit more expensive, lightning models. In this paradigm, rays of light are simulated in the scene and can be used to transfer energy between surfaces, naturally accounting for effects such as reflections, indirect illumination, shadows, and caustics. During simulation, each ray sent must determine the closest visible surface along its path in order to continue its interaction with the scene. The high cost of this method comes from the large number of rays that need to be traced randomly throughout the scene in order to converge to a visually noise-free image. An illustration showing the conceptual difference between rasterization and ray tracing is shown in Figure 4.

In physically based rendering, materials are typically described by the bidirectional reflectance distribution function (BRDF) [25] and many different material models exist in this framework. The focus of this thesis is not the subject of materials, but

how to quickly and efficiently answer questions on the form, *"Which triangle is visible in this pixel?"* or *"Which surface lies in front of this ray?"*. When time is added to the equation, the question becomes more complex and involves over which interval(s) in time a triangle lies in front of a pixel or ray.

Rendering is a sampling problem at its core and generating images of higher quality usually involves taking more samples (in addition to improving the sampling strategy itself). Thus, fast visibility queries for each sample is paramount to the performance of any rendering system.

2 Measuring Image Errors

The quality of visibility samples will usually show in the final rendered images in obvious ways. Faster algorithms often introduce approximations which can lead to errors such as noise, aliasing, and inaccuracies. This is, however, not always the case. When proposing a new algorithm with approximations, it is therefore necessary to compare both rendering time and image quality to previous work in the field. In order to evaluate image quality in an objective manner, a ground truth image is rendered using some brute force method, that is known to provide a correct result. These runs can take a very long time to complete. The alternative algorithms also render the same image, but preferably much more quickly. The result is a set of images, one for each algorithm, and a reference image that is assumed to be perfect. In order to quantify the likeness between one image and the reference, a formula for computing the error between them is necessary. While several metrics exist, this thesis makes use of the metric outlined below.

The mean square error (MSE) is first computed by squaring the difference for each pixel between the reference image and the image to compute the error for, normalized to the total number of pixels, which can be expressed as

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \big(P(i,j) - P_{ref}(i,j)\big)^2, \tag{1}$$

where m,n indicate the image dimensions and P,P_{ref} give the pixel value at a location for the algorithm and reference respectively. If the image contains multiple color channels (commonly red, green, and blue), the MSE is computed for each of them and the result is averaged together.

In order to arrive at a more pleasant metric for image errors, the MSE is normalized by inverting and multiplying by the squared maximum intensity of the image (255^2 in an 8-bit image). The result is a ratio that gets higher as the two images gets more alike. By computing the 10-logarithm of this value, and multiplying by 10, the resulting unit is decibel (dB). This measurement is known as peak signal-to-noise ratio (PSNR) and is commonly used to benchmark lossy image compression algorithms. The resulting formula can thus be written as

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right). \quad (2)$$

When using PSNR as a metric, it is important to note that a single PSNR value cannot account for all aspects of image quality. For example, an image could be perfect in most places but exhibit unreasonable artifacts in others. Therefore, when making comparisons, it is important to inspect images and note any obvious artifacts manually. In addition, since the difficulty of a scene can impact the absolute PSNR value, it is not meaningful to compare values between different rendered scenes.

PSNR values alone are often not so interesting since many slow algorithms can generate high quality images given enough time. However, by aggregating the time it takes to generate an image with the achieved image quality, meaningful conclusions can often be made. A strong case can generally be made for an algorithm that is both faster and of equal or higher quality in a variety of situations.

The structural similarity index (SSIM) is another approach to measuring similarity between images that have been claimed to be consistent with human perception [38]. While such metrics can be very useful, SSIM may accept systematic changes to an image that preserves structure, such as contrast-stretching and mean-shifting the color channels, which is usually not desired in visibility computations. In addition to being more difficult to compute, a more complex algorithm can also make the resulting score harder to reason about.

3 Hardware Platforms

Different hardware platforms exist to support rendering algorithms. The two major ones are central processing units (CPUs) and graphics processing units (GPUs). CPUs are capable of running general purpose instructions to support a wide range of programs, while GPUs are often a bit more restricted in flexibility to support a more parallel programming model. While CPUs are optimized for a few, possibly different, parallel instruction streams, GPUs are optimized for having thousands of data items executing a relatively small kernel in parallel. In addition, GPUs often include fixed function hardware specifically targeted at graphics workloads. In this work, we make extensive use of their particular strengths and will thus briefly introduce them before presenting the contributions of this thesis.

3.1 CPU

A CPU is central to most computing platforms and is used in most systems running general purpose software. While they come in various configurations and sizes, we focus on modern desktop/laptop CPUs in this work. In particular, the focus is on modern x86 processors [20].

Several CPU cores per chip have been an increasing trend in recent years. In the best case, this leads to a linear scaling in performance with respect to the number of available cores. However, Amdahl's law [21] tells us that the possible speedup will be limited by single threaded portions of the application as the number of cores increase, so more cores is only a partial solution. Key challenges for multi-core CPUs involve cache coherence and synchronization. At the software level, algorithms need to be carefully designed to be split into independent parts that can be executed in parallel. While this is a natural fit to some problems, others require elaborate schemes to run efficiently.

While thread level parallelism is suitable for coarse grained task-based distribution of work, a finer grained form of parallelism exists at the instruction level. By performing the same computation on several data items using single instruction multiple data (SIMD) execution, a vector of results can be computed in the same time it takes to otherwise calculate a single value. The extra performance scales linearly with SIMD width for suitable algorithms and is very power efficient. Current trends indicate that SIMD width will continue to increase in the future. For example, contemporary CPUs have a SIMD width of 256 bits, and this will likely be extended to 512 bits in the near future. The many-core Xeon Phi architecture already features 512-bit SIMD, and most GPUs have a width of 1024 or 2048 bits. Algorithms need to have low instruction level divergence in order to efficiently utilize increasing SIMD width. For example, if some parts of the vector wants to perform a different computation in some cases, efficiency will be lowered for those cases.

CPUs have also developed a variety of techniques to improve performance of scalar instructions without changing existing programs. By rearranging instructions, they are able to find inherent parallelism in a single instruction stream [21]. Superscalar execution makes it possible to execute multiple instructions simultaneously on different functional units and pipelining is used to issue new instructions each clock cycle. Each pipelined instruction can have different *latency*, which is typically longer than a clock cycle. The strength of this setup is that high throughput can be achieved with independent instructions. When the CPU is unable to issue a new instruction, e.g., due to dependencies between instructions, a bubble is created in the pipeline that does not do any real work. To counteract this, instruction scheduling is performed to try to avoid these issues, both when compiling a program to machine code, and as stated above, by the actual hardware during execution. However, without enough independent instructions, this scheduling will not succeed in eliminating the pipeline bubbles. To take full advantage of a modern CPU, algorithms should therefore strive to provide enough parallel work within a single instruction stream.

Another key challenge is that when compute capabilities increase, more pressure is put on the underlying memory subsystem. To reduce latency and bandwidth to main memory, CPUs are equipped with a large cache hierarchy in combination with sophisticated hardware prefetchers that detect memory access patterns, and prefetch memory into a cache before it is needed. Nonetheless, an instruction that

incurs a cache miss, and have to go out and accesses main memory, have a latency of several hundreds of cycles. Because of this, data layout and access patterns are perhaps the most important design aspects of algorithms on modern CPUs.

3.2 GPU

GPUs were historically dedicated for real-time graphics workloads. Initially they were limited to fixed function hardware without programmability that included two-dimensional rasterization with a fixed transformation pipeline and fixed function fragment shading with support for textures and per-vertex light computations [1]. Visibility among rasterized fragments was (and still is) handled using a z-buffer [7, 34]. In the case of transparency, composition is performed using blending hardware. With time, support for multi-texturing and register combiners were added to improve the flexibility of the pipeline, ultimately ending up in user-defined shader programs at different stages of the pipeline. The first shader stages to be added were a vertex shader stage, that controls how vertices are transformed before rasterization, and a fragment shader stage, that computes the color of each fragment after rasterization [29]. Rasterization, texture filtering, and blending of fragments is still mostly fixed function today.

Other shader stages have been added with time and now there is a geometry shader for generating or changing geometry (such as triangles), as well as shaders for controlling tessellation (hull shader and domain shader). As support for increasingly complex shader programs was realized, the set of possible GPU applications expanded. With the advent of compute centric platforms, such as CUDA [31] and OpenCL [27], the GPU was transformed into a platform for general compute problems on its highly parallel shader cores. In our work, we make use of the existing graphics pipeline for rendering and utilize compute programs to efficiently implement some of our algorithms.

Similarly to modern CPUs, the shader cores in a GPU are SIMD machines where the same instruction is performed on multiple data items during program execution. The fundamental design differences between a CPU and a GPU is how they handle memory access latency and how they execute parallel code. A CPU is optimized for a few parallel threads of execution, trying to minimize memory latency using large caches. GPUs are instead designed to tolerate long latency and focus on high throughput. This is accomplished by keeping hundreds or thousands of threads active at any instance in time. When a high latency operation, such as a read from main memory, is initiated, the shader core simply switches thread and continues to compute. If enough threads are active, the memory fetch will be done by the time execution resumes the originally suspended thread. While caches do not serve the same purpose for a GPU, they are still present to reduce memory bandwidth usage.

Since the number of active threads on a GPU is very large, each thread can only use a limited amount of resources. For example, the register file is divided among all threads running on a single processor on the GPU, meaning that a single thread

cannot use more than 64-128 registers without reducing parallelism, and thus loosing efficiency. The same is true for shared caches and fast on-chip memory that can be used for synchronization.

Although a GPU needs to keep many threads active in order to tolerate latency, most threads are idle at any given moment and do not progress forward. For this reason, traditional synchronization techniques used for multi-core CPUs, such as mutexes, become unusable and would result in deadlock. Current programming models for GPUs are inherently data parallel and allow the user to believe that scalar code is being written. The premise is that the code will then run for many data items in parallel. This parallelism comes from both thread level parallelism and SIMD execution. Behind the scenes, the compiler generates SIMD instructions so that a SIMD-width of data items essentially work in lockstep.[1] This realization is crucial for optimizing many applications as data items within a SIMD-width can have very efficient synchronization and communication. It is also important that lanes within the same SIMD-width do not diverge, i.e., if there is control flow in the program, all SIMD-lanes should take the same branches. Otherwise efficiency will be reduced, much like how SIMD behaves for CPUs. In fact, as CPUs are getting wider SIMD, higher computational throughput and more registers, a software scheduling of many independent CPU instructions can effectively hide memory latency to similar effect as how a GPU hides latency using threads. It can therefore be argued that the two platforms are converging in program behavior. However, for real-time graphics, the additional fixed function hardware in GPUs have proven hard to match with a pure software implementation of similar functionality.

3.3 Shared Memory

As small form factors for computers and low power consumption have become increasingly important, GPUs and CPUs have been integrated on the same chip in order to improve performance, power consumption, and reduce cost. These architectures typically share the same memory subsystem, which presents unique challenges and opportunities for work sharing. Currently these architectures are common in cell phones, tablets, and increasingly so in laptop computers.

4 Contributions and Methodology

This thesis presents a collection of algorithms that can be used to speed up or improve the quality of visibility queries in graphics. Here, we will review my main contributions and connect back to the introduction. More details will be presented in the following sections. I will also discuss my part in co-authored papers, and would like to start by saying that my main supervisor, Tomas Akenine-Möller,

[1] In GPU parlance, the term "thread" is often used to refer to individual SIMD lanes, which is confusing when compared to a CPU, where a single thread supports multiple SIMD lanes.

has been actively involved in all papers presented in this thesis, including algorithmic design, illustrations, and text authoring. My work specifically target fast algorithms for rendering on currently available hardware, i.e., no hardware simulations have been performed. As such, our main evaluation methods have involved rendering time, typically milliseconds per frame, and image quality, using PSNR. In some cases, we have looked at memory requirements and memory bandwidth usage.

My early work was focused on alternative methods to stochastic point sampling for higher-dimensional visibility problems and difficult geometry when using rasterization. Paper I presents an approach for semi-analytical visibility coverage in time over spatial line samples. This allowed for noise- and alias-free rendering of images with motion blur, such as the one shown in Figure 1. In addition, it was faster than our stochastic software rasterizer while also generating higher quality images when measuring PSNR. In this paper, I focused on the overall system design and implemented the visibility engine and shading system. The first author, Carl Johan Gribel, focused on the tessellated surface between a moving triangle and a line sample plane. Paper II introduces a new way to render anti-aliased thin curves, such as hair. In this paper, I once again focused on the system design, and worked on the efficient GPU implementation. I also developed the heap based interval resolve procedure to quickly determine visible curve segments. Carl Johan focused on generating high-quality intersection points between a Bézier curve and the line sample plane.

After the first two papers, my focus shifted from analytical line-based algorithms towards improving methods for performing traditional point sampled visibility. Paper III targets ray tracing and discusses a way to perform arbitrarily ordered ray traversal of a binary tree without a stack, which can be beneficial when storing a stack is expensive. Paper IV introduces a way to use shared memory between CPU and GPU to perform anti-aliasing on idle CPU cores in a rasterization based renderer. The GPU rendered the scene using a single point sample per pixel in a traditional hardware accelerated pipeline. In parallel with the GPU, the CPU detected difficult pixels and used an optimized software rasterizer to render only those pixels in high quality, while using resources generated by the GPU in shared memory. Paper V discusses how large packets of rays can be traversed simultaneously against a bounding volume hierarchy, in order to increase the available data parallelism. The resulting algorithm was able to improve the performance of ray traversal by up to 50% on a modern laptop CPU compared to a set of ray traversal kernels that can be considered state of the art.

5 Visibility Determination

Visibility determination is the process of computing overlap between one primitive with respect to another in some space. The answer can be as simple as a boolean indicating whether a hit between a ray and a triangle occurred or as complex as a

three-dimensional surface representing the overlap between a plane and a moving triangle. In this section, I will discuss visibility in different rendering algorithms, and how they relate to my work. In particular, two types of rasterization will be introduced, and the related subject of anti-aliasing, followed by a discussion about ray tracing.

5.1 Two-Dimensional Rasterization

The simplest useable camera model is that of a pinhole camera. It has no lens area, meaning that depth of field effects can be ignored. In addition, the shutter interval is infinitely short, resulting in that no time-dependent effects, such as motion blur, can happen. Each location on the camera sensor will thus correspond to a single direction of light from the scene, with all light coming from the same instance in time (assuming light is modelled as traveling infinitely fast). While these assumptions my sound unrealistic, they are excellent for real-time games where it is hard to know where the player's eyes are focused for proper depth of field during play. It can, however, be used to great effect to guide the player's attention during cinematic sequences. Motion blur is generally something favorable but its effect is mostly limited to fast moving objects (relative to the camera), which is why it can be reasonable to ignore or approximate it in real-time rendering. The real benefit of these assumptions is that they allow rendering with a computationally efficient algorithm.

The basic goal of two-dimensional rasterization is to determine what lies in front of each pixel from the camera's point of view. Instead of having a camera, like in the real world, where a sensor gathers light from the environment, light paths are modelled in reverse. One way to think about this is to trace straight lines, i.e., rays, from the camera sensor out into the scene and what each ray hits in the scene determines the color at that point on the sensor. The pin hole camera model guarantees that all rays are directed coherently and allows the problem to be reformulated as a rasterization problem. Instead of tracing rays in three-dimensional space, an equivalent result can be achieved by projecting all triangles to the image plane and perform point vs. triangle tests in two dimensions [32]. This concept was shown in the upper part of Figure 4 in the introduction. The following paragraphs will introduce a typical modern two-dimensional rasterization pipeline with programmable vertex and fragment shading. This relates to our work on thin curve rendering in Paper II and, even more so, to our work in Paper IV where an optimized software implementation of a complete two-dimensional rasterization pipeline was developed. A subset of the problem introduced in Paper I, in particular the visibility resolve in ltz-space, can be directly mapped to a two-dimensional rasterization problem. Geometry shaders and tessellation are intentionally left out of the discussion as my thesis have made no use of geometry shaders and very little use of tessellation (it can be found as competing algorithm in Paper II).

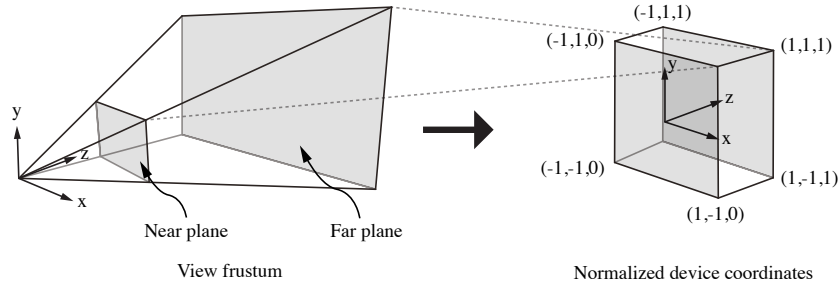


Figure 5: *A projection matrix defines a frustum with a near and a far plane (left). Anything outside the frustum is not drawn. Each point inside the frustum gets mapped into a box (right). The coordinates in this space is commonly referred to as normalized device coordinates. Note that the coordinate system used here is only an example and certain implementations my have different conventions. For example, the z-axis in the frustum is commonly directed the other way and the z-axis in normalized device coordinates may span* $[-1, 1]$ *instead of* $[0, 1]$.

Overview

The input to a typical rasterization pipeline is a list of vertices and a list of indices organized in batches, called *draw calls*. The result of rendering operations are stored in an offscreen buffer known as the frame buffer, which ultimately contains the final image once rendering is complete. Each draw call will have associated with it a vertex shader and a fragment shader. The vertex shader transforms the individual vertices before rasterization, including projecting them to the image plane, while the fragment shader determines the color that ultimately ends up in the frame buffer. Three consecutive indices make up a triangle by addressing into the list of vertices in order to facilitate reuse when vertices are shared by multiple triangles. Each vertex describes the position of the vertex as well as other possible attributes that are needed during vertex shading, e.g., visual attributes to be passed on to the fragment shader when computing the color. Examples of vertex attributes are surface normals and per-vertex color information.

Vertex Transform and Clipping

The first step in processing a draw call is to run the vertex shader on all its vertices. This shader will transform vertices from object space, in which the vertices are defined, to the image plane. A three-dimensional camera with perspective represents a non-affine transform in three-dimensional space. In order to express a projection using a linear transform, the transformation pipeline is based on four-dimensional *homogeneous coordinates* that introduces an additional component, w. In this homogeneous coordinate space, a point scaled by a constant represents the same point. This system allows a projective transform to be defined as a 4×4

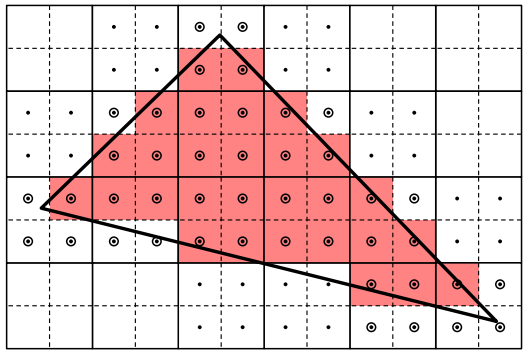


Figure 6: *A triangle rasterized using 1 point sample per pixel. Hierarchical rasterization tests each* 2×2 *pixel quad and rejects those trivially outside. Otherwise, inside and possibly depth tests are performed for each sample. Shading is usually performed at the center of the pixel for quads where at least one sample overlaps the triangle. In practice, each quad may be bigger than* 2×2 *pixels.*

matrix where the projected position is given by scaling the vector by $1/w$. The camera frustum defines a near and far plane along its z-component and the region between them are mapped into a box-shaped three-dimensional space, sometimes called normalized device coordinates, through the division by w (see Figure 5).

Before division by w, care must be taken so that only triangle regions that really should be visible are projected to the screen. This intermediate space is known as *clip space*. Part of that condition requires that visible regions should be inside the near and far plane of the camera frustum. To ensure that this is the case, triangles beyond those planes will be discarded, and triangles straddling them will be clipped. In general, triangles are clipped so that all points fulfill $0 < z < w, w > 0$, when normalized device coordinates have a z range of $[0,1]$. Triangles that lie between the near and far planes can also stretch far outside the visible camera frustum in the x- and y-directions. Generally, a rasterizer can efficiently skip parts of triangles that are outside the screen, but there is usually a fixed precision that the rasterizer supports. If triangles extend too far outside the frustum, this precision may overflow. In order to handle this, additional clip planes are added outside the frustum to guarantee that the rasterization precision is enough.

Rasterization

After clipping and projecting triangles, the next task is determining which pixels each triangle overlaps. The x and y coordinates in normalized device coordinates can be directly mapped to image pixels. The problem thus becomes determining overlap between pixels and two-dimensional triangles. There are multiple ways to do this.

The conventional method of rasterization involves taking point samples to approximate visibility over a pixel. The simplest method simply takes a single sample at the center of a pixel. While being very fast, this may lead to low quality images with aliasing artifacts. Here we assume this method is being used. More elaborate methods will be discussed in the anti-aliasing section below.

One approach to determine visibility over pixels involves rendering scan lines from the leftmost edge to the rightmost edge, one row at a time. A more parallel algorithm, usually employed by graphics hardware, involves computing half space equations for each edge of the triangle that together determine if a point sample (usually the pixel center) is overlapped [32]. By extending the half space equations, whole regions of pixels can be tested at a time [2], which forms the basis for hierarchical rasterization. The process of hierarchical rasterization is illustrated in Figure 6.

A modern rasterization pipeline rasterizes a potentially visible triangle into one or more quad fragments. Fragments may then be depth tested, as described below, and, if they pass the depth test, proceed to be shaded by a fragment shader. Finally, visible fragments are composited to the frame buffer using a blend operation. These steps are described in detail below.

Hidden Surface Removal / Depth Testing

In cases where multiple triangles cover the same pixel, a way is needed to determine which triangle is in front of the others. One way, known as the painter's algorithm, is to sort the triangles so that the closest triangle is drawn last, and thus overwrites the previous triangles rendered before it. Sometimes, however, a sorted order cannot be established for a set of triangles, such as three mutually overlapping triangles or two intersecting triangles. In this case, triangles must be clipped against each other, usually by building a binary space partitioning (BSP) tree. BSP trees were common in early real-time graphics applications and could be generated offline for static geometry.

A conceptually simpler approach, known as z-buffering [7, 34], involves storing the closest depth value for each point sample and discarding samples that are farther away. The depth value for a sample can be interpolated from the z-values of the individual vertices that make up a triangle in normalized device coordinates. The depth test often allows for efficient culling of fragments before fragment shading. At the time when the algorithm was invented, it was considered expensive because of the memory requirements [7]. Since memory is inexpensive today, it has become the de facto standard for resolving depth order between opaque fragments.

Fragment Shading

Fragment shading is often one of the most expensive operations in rendering and it determines the color of fragments that will be composited to the frame buffer. A fragment shader program computes everything related to a fragment's appearance,

which usually involves the geometrical properties such as normal and tangent, surface properties such as roughness and color, and for each light source, light properties such as position, fall-off, and shadow map. More generally, fragment shading involves interpolating visual attributes from vertices, looking up filtered data in *textures*, and performing computations to arrive at a final color.

Two-dimensional textures are rectangular grids of pixels, much like the image that we are trying to generate, that can be mapped onto triangles to add visual richness. These textures can be authored offline or be generated by a previous rendering pass. In the context of textures, pixels are usually referred to as *texels*. A two-dimensional texture lookup in a shader passes a set of texture coordinates to the texture unit and receives a filtered sample. Different filtering modes exists. The simplest filtering mode simply returns the color of the texel closest to the coordinate. This can, however, lead to a pixelated appearance if a texture is magnified. Bilinear filtering will, on the other hand, look up the four texels around the coordinate, and then perform linear interpolation between them, which leads to a smoother appearance.

Both of these methods will, however, perform poorly when an image is reduced in size. During minification, the sampling frequency of the texture gets too low, typically resulting in aliasing, and flickering during movement. One way of reducing the issue is to take multiple texture samples over the footprint of an image pixel. This is, however, a very expensive way to solve the problem as a pixel can potentially cover many texels in a texture. The typical way to solve the problem in real-time graphics is to provide a set of pre-filtered versions of the texture such that each additional image contains the previous image filtered to half the resolution. A texture lookup may then select a texture in the set with an appropriate sampling frequency. This technique is known as mip-mapping and was introduced by Williams in 1983 [39]. A sharp transition between mip-map levels can be avoided by performing two lookups at different levels and then interpolate between their results.

In order to determine a suitable mip-map level, the footprint of the pixel in texture space is needed. This footprint depends on the rate of change of the coordinates used to look up in the texture with respect to the pixel position on the image being rendered. As such, if the texture coordinates are given by (u,v), and the pixel position is described by (x,y), the quantities that are needed are $\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, and $\frac{dv}{dy}$. It is possible to compute these derivatives analytically for attributes that vary over a triangle, but if the result from one texture lookup is used to offset the lookup of another texture, analytic derivatives get problematic. This is one of the reasons that fragment shading typically is performed in quads. Quads of four or more pixels allow the shader to efficiently map shading work to a SIMD machine, which includes both CPUs and GPUs as described in Section 3. The additional benefit of quad shading is that derivates are very easy to approximate. If the SIMD machine provides some simple horizontal operations over its SIMD lanes, the derivatives of a quantity can be approximated using a simple finite difference. It is, however, worth noting that diverging control flow among SIMD lanes may introduce severe

errors in the derivative computations as some lanes may be masked out. While convenient, quad fragment shading can lead to problems when triangles become very small. If a single triangle only covers one or two pixels on average, the amount of unnecessary shading gets expensive and increases as larger shading quads are used. This problem and possible solutions have been explored in recent work [8, 12].

A problem with isotropic mip-mapping is the fact that the pixel footprint may not be a square in texture space. Thus, the user is forced to choose between over-blurring and aliasing. Methods to address these issues involves anisotropic texture filtering, where one approach is to take multiple texture samples at a suitable level in the mip hierarchy.

In addition to two-dimensional textures, shaders often support one- and three-dimensional textures as well. The above filtering methods typically apply to them in an analogous way.

Blending

Once the color of a fragment has been determined through shading, multiple fragments may overlap the same pixel and must be composited into a final image. Often fragments are opaque, and in this case, the colors of the front-most samples determine the final color of the image pixel. However, if multiple fragments overlap and they are transparent, their values must be combined according to their properties. This is usually accomplished by conceptually writing a fragment at a time to the frame buffer, while blending the result with the previous value already in the frame buffer. Graphics hardware typically include fixed function hardware for performing blending between fragments and the content of the frame buffer. How this hardware works ties into how ordering is performed between fragments.

Often the result of blending, and even depth testing, depends on the order in which fragments are processed. In order to guarantee deterministic behavior for rendering, fragments from triangles must generally be processed in the order they were submitted to the rasterization pipeline. A parallel rasterization pipeline can be characterized depending on where ordering is enforced [30]. Possibilities include sort-first, sort-middle, and sort-last. The systems most actively in use today are sort-middle and sort-last architectures.

In sort-middle, order is enforced after geometry processing but before rasterization and fragment processing. This is accomplished by dividing the screen into multiple bins (also called tiles) and distributing triangles to bins they overlap after projection. Each bin can then be rendered in parallel. Such architectures are currently common in mobile devices and other low power systems. Advantages include that all blending and depth tests can be performed in fast on-chip memory, alleviating the need to go out to external memory when accessing the depth and frame buffers. Downsides include bin spread when a triangle overlaps multiple bins. The buffer for each bin may also overflow, which makes it debatable whether sort-middle architectures can be efficient when geometry is highly detailed, such as

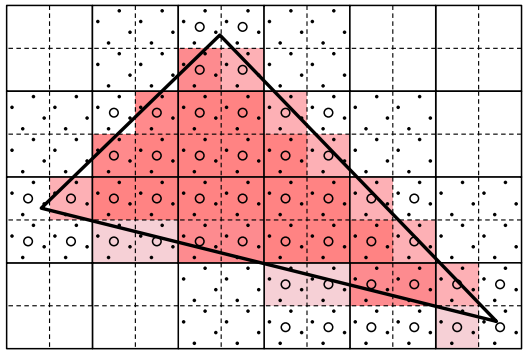


Figure 7: *A triangle rasterized using 4 visibility and one shading sample per pixel (4× MSAA). Four inside and depth tests are here performed per pixel, and the covered samples are used to blend shading between the background and the triangle. While not completely eliminated, aliasing along the triangle edges have noticeably improved compared to Figure 6.*

when making heavy use of tessellation. When these architectures perform blending, a single graphics core is typically working on a tile. It is thus easy to perform blending in software, and these platforms often provide support for programmable blending for exactly this reason.

Sort-last architectures are usually found in high-end graphics processors. Here, everything is completely parallel up to fragment blending. Fragments are therefore generated and shaded out of order. At the very end of the pipeline, large reorder queues are used to ensure that blending happens in order. Blending is here performed using fixed function hardware and is thus limited to a set of predefined blend operations.

Anti-Aliasing

The simplest form of rasterization checks if the center of a pixel is contained within a triangle. This can lead to aliasing artifacts along edges of triangles and have been mentioned as one of the biggest problems in real-time graphics [5]. Higher quality rasterization exists that take the fractional coverage of the triangle into account. The simplest form of anti-aliasing is super-sampling, where multiple samples are taken inside each pixel. Each sample can hit different triangles and, when weighted together, they provide a better approximation of what the entire pixel covers. Super-sampling is, however, prohibitively expensive since it requires not only more visibility tests within a pixel and additional depth buffer storage for ordering, it will also shade each sample using the fragment shader. The most prominent approach in real-time rendering is called multi-sample anti-aliasing (MSAA). MSAA is an attempt to reduce the cost of anti-aliasing, compared to full super-

sampling. This is accomplished by decoupling visibility samples from shading samples. While multiple visibility samples and depth tests are performed, only a single shading sample is taken for each triangle overlapping a pixel. That is, if all visibility samples in a pixel overlap a single triangle, a single shading sample will be taken. This is likely the most common case for scenes with large to moderately sized triangles. Along edges and other aliased parts, multiple shading samples will be weighted together. An illustration of rasterization with MSAA is shown in Figure 7. Recently, MSAA was extended to support shading less than once per pixel, which can be beneficial for high density displays [37].

Other, more elaborate, ways to solve the aliasing problem include trying to analytically solve for coverage instead of taking multiple samples. In Paper II, we analytically solve for coverage over spatial line samples in order to render thin curves in high quality. Our software GPU implementation was comparable in quality and speed to a hardware accelerated pipeline with tessellation and fixed function rasterization with MSAA.

In Paper IV, our A^4 algorithm attempts to reduce the overhead of MSAA by only focusing on pixels that are likely aliased. While the GPU renders a scene using a single sample per pixel, the CPU detects silhouette edges and proceeds to render them in high quality in parallel with the GPU. Shared memory allowed for data generated by the GPU to be easily accessed by the CPU during rendering.

5.2 Higher-Dimensional Rasterization

In two-dimensional rasterization, visibility is determined by sampling over the image x and y coordinates. If, like in the real world, the camera captures light over a finite time interval, objects can move as light reaches the image sensor. This time interval is usually referred to as exposure time or shutter interval. With this setup, fast moving objects will appear blurred as they move across pixels in the image. It does, however, introduce complexity to our rendering model. First, an additional dimension, time t, is added. This means that we now have a three-dimensional sampling problem over x, y, and t. The second complexity is how motion of objects are modelled. Here, multiple options exist. The simplest method is to consider linear vertex motion, which means that all vertices store two positions: one position $\mathbf{q}$ at the start of the shutter interval and one at the end of it, $\mathbf{r}$. Positions of a vertex at time t is then given by the linear combination $\mathbf{p}(t) = (1-t)\mathbf{q} + t\mathbf{r}$, when $0 \leq t \leq 1$. This approach is illustrated in Figure 8. By looking carefully at the figure, a problem with linear motion becomes apparent. The triangle may appear to shrink during motion, even though it has the same size at the start and the end of the shutter interval. The problem generally gets worse as the amount of motion increases and it is particularly visible when objects are rotated by large angles. In this case, entire objects composed of many triangles can appear to shrink. Ways to remedy these problems include approximating motion using higher order polynomials [17]. In this work, however, the focus has been on linear motion with the assumption that the amount of motion is relatively small, or that large motion

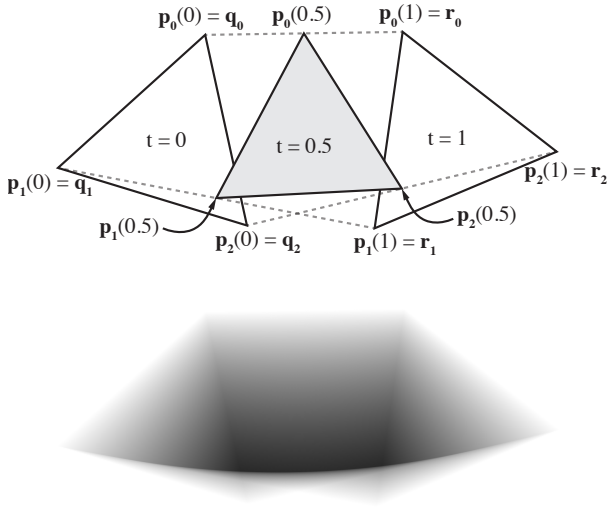


Figure 8: *Triangle with per-vertex linear motion. Two positions,* **q** *and* **r***, are defined per vertex and they are linearly interpolated to get the triangle at a specific time. The result of three-dimensional (x,y,t) rasterization of the triangle (with black color) is shown at the bottom. Note that the triangle appears smaller at $t = 0.5$. This can be seen as a limitation of this method.*

has been refined using piecewise linear motion.

Rasterization can also be extended to even more dimensions. In order to model how depth of field works in a camera, the area of the lens must be included in the sampling space. A common model is to parameterize the lens using u and v. Together with time, the resulting problem is referred to as five-dimensional rasterization as the sampling space now has five dimensions, namely, x, y, u, v, t. While five-dimensional rasterization is a very interesting research topic, it is outside the scope of this thesis. The work related to ray tracing is, however, directly applicable to these higher-dimensional sampling problems (see Section 5.3 for details).

Specific methods for performing three-dimensional rasterization is outlined below, including a brief introduction to my work in semi-analytical three-dimensional rasterization.

Stochastic Point Sampling

Five-dimensional rasterization can be realized by randomizing point samples over the five-dimensional domain. In 1987, Cook et al. [9] introduced the REYES rendering system using rasterization of motion blur and depth of field with stochastic point sampling. Akenine-Möller et al. [4] introduced the concept of stochastic

rasterization using time continuous triangles. Other work focus on small triangles (micro-polygons) with motion blur and depth of field [13]. Ragan-Kelley et al. [33] present a method to decouple shading from visibility samples with motion blur and depth of field, which is similar to what multi-sample anti-aliasing does for *xy*-samples.

Stochastic three-dimensional rasterization using time-continuous triangles was used as competing algorithm in Paper I. In this paper, we also introduced a simple shading cache to decouple point sampled shading from visibility across all triangles in the scene. It tried to achieve a uniform number of shading samples per image pixel by determining required resolution using screen space differentials of mapping coordinates. Shading storage of appropriate resolution was lazily allocated during shading.

Analytical Methods

Gribel et al. [16] used spatial point samples and analytical visibility in time. A list of time intervals with depth and color information was generated for each spatial point sample and was resolved analytically to get the final color. This resulted in noise-free motion blur, but aliased edges on static geometry. Earlier, Jones and Perry used spatial line samples to get high-quality anti-aliasing for static geometry [24].

In Paper I, visibility was analytical over spatial line samples and in time, thus combining the ideas of Gribel et al. with the work of Jones and Perry. Moving triangles intersecting line sample planes traced three-dimensional surfaces in ltz-space, where l is the dimension along the line. These surfaces were approximated using triangles and visibility was resolved to get the front-most triangles over space and time. Each line thus essentially became a two-dimensional rasterization problem with an orthographic projection. To get analytical visibility in ltz-space, we used a BSP-tree to get depth sorted polygons and applied a clipping algorithm introduced by Catmull in 1978 [6]. Here, each edge clips the existing polygons into two groups recursively in order to compute the exact polygon coverage. The term "semi-analytical visibility" used in the paper comes from two approximations. First, the intersection surfaces between triangles and a line sample plane are tessellated. Second, a full analytical solution over the *xy*-plane was not achieved since analytical visibility was restricted to lines. Still, practically noise- and alias-free images were achieved with favorable performance.

Additional work on line sampled visibility include the work on depth of field by Tzeng et al. [36] and the analysis of line sample patterns by Sun et al. [35].

5.3 Ray Tracing

Ray tracing is typically more flexible than rasterization. Instead of projecting geometry to the screen to reduce the dimensionality of the problem, rays are traced

and tested against the scene in three dimensions. Each ray can typically be traced independently of others and may depend on the result of previously traced rays. It is thus quite easy to arrive at results similar to full five-dimensional rasterization by generating rays from the camera using different (x, y, u, v, t)-samples. It is also possible to implement adaptive sampling schemes that emit more rays in difficult regions of the image. The most attractive feature of ray tracing is, however, that it can accomplish much more than primary visibility from the camera. Once a camera ray hits a surface, the incoming radiance can be estimated at that point by sampling its surrounding. This sampling is typically performed by sending one or more rays. The process can happen recursively and allows the renderer to arrive at a global illumination solution where, e.g., diffuse interreflections and shadows are accurately captured.

There exists multiple algorithms for computing global illumination using ray tracing. Path tracing [25] is one of the simpler ones that lets each initial ray from the camera continue its path in the scene randomly until a termination criteria is met. This method will, however, have trouble capturing pure specular light paths from small light sources, such as caustics from a point light, because a ray has very low (or even zero) probability to sample a small point light directly. Bidirectional path tracing [26] allows light paths to be sent from both the camera and light sources in order to better handle difficult light conditions, but still cannot handle pure specular light paths. Algorithms such as photon mapping [23] and progressive photon mapping [18] handle this case by shooting rays from light sources in one pass and then issuing an extra pass to gather them from the camera. Recent work in the field include ways to combine features of both bidirectional path tracing and photon mapping into more robust methods [14, 19].

While the focus of this thesis is not on the topic of light transport algorithms, what is common between all these algorithms is that rays are generated and traced. Testing a ray against all primitives in a scene is typically prohibitively expensive since a scene can contain many millions of triangles. Fortunately, the problem can be made manageable using a spatial data structure to speed up the query. Using a spatial data structure, it is possible to quickly prune parts of the scene not overlapping a ray in a hierarchical manner. The hierarchical pruning is commonly referred to as *traversal*. While many different spatial data structures exist [10], the bounding volume hierarchy (BVH) is currently very popular due to features such as ease of implementation, predictable memory requirements, and high performance. In this thesis, the contributions to ray tracing focus on algorithms for traversing a BVH for efficient ray tracing. In the following section, a brief introduction to the BVH and its construction is given, followed by an overview of our novel traversal algorithms.

BVH Construction and Traversal

A BVH arranges geometry in a tree structure where each leaf node contains the primitives (e.g., triangles) to be rendered. Each internal node in the tree stores a

bounding volume that encloses all geometry in its subtree. It is thus possible to skip an entire subtree if a ray does not intersect the bounding volume. There are multiple choices for bounding volumes, such as spheres, axis-aligned bounding boxes, oriented bounding boxes, and even general convex polyhedra. More complex bounding volumes are generally tighter, but also require more expensive computations when determining overlap with a ray. A good tradeoff between culling efficiency and computational cost is the axis-aligned bounding box, and as such, it is currently the most popular bounding volume. Each internal node often has two children. However, a so called multi-BVH [11], where each internal node has more than two children (such as 4 or 8), can be useful to support SIMD execution when traversing a single ray against a BVH. In this case, all bounding volumes of the children can be tested simultaneously using vector instructions.

A BVH is generally constructed by optimizing a heuristic cost metric, such as the surface area of all bounding volumes [15, 28]. The algorithm can proceed in a bottom-up, top-down, or insertion-based manner. Bottom-up construction starts with all nodes as leaf nodes and tries to merge the best pairs. Top-down builders partitions all primitives recursively into subtrees. Insertion-based methods starts with an empty tree and iteratively adds the best node to the tree using some heuristic.

Independent of how a BVH is built, traversal works in the same way. In conventional ray traversal, a single ray is traversed against a BVH at a time and a stack is used to keep track of nodes that are still to be traversed. When a binary BVH is traversed, and a ray intersects both children of a node, a choice needs to be made which node to start traversing, and which node to postpone traversal of by pushing that node to the traversal stack. This choice can have great impact on the resulting rendering performance. If the closest hit is needed, such as for general light paths, it is natural to start with the closest node first. In case a hit is registered in the closer node, the farther node can often be culled from processing. For shadow rays, when a simple boolean indicating whether a hit occurred or not is required, an efficient heuristic involves the probability for a ray to be occluded inside a node [22]. In Paper III, a method to perform single ray traversal of a binary tree without a stack was introduced. It was found that the traversal state could be stored in two integers with only one bit of storage per level of the tree. In contrast to previous stackless traversal approaches, our method is able to support arbitrary traversal order.

In Paper V, we introduced an efficient way to traverse large packets of rays against a BVH that allowed wide SIMD units to be efficiently utilized. The goal was to provide enough parallel work, even for incoherent rays. As such, each ray was allowed to choose its optimal traversal order. Rays taking the same path in the tree was then grouped and executed in a streaming manner. In contrast to previous approaches leveraging large ray packets, our method allowed for dynamic traversal order and high efficiency for highly incoherent rays.

6 Conclusions and Future Work

The area of visibility queries is certainly not fully explored. In this thesis, different techniques targeting various visibility problems have been proposed. Motion blurred triangles were rendered on a multi-core CPU with semi-analytical coverage over time and spatial line samples. Thin curves were rasterized in high quality on a graphics processor using spatial line samples and curve-specific intersection tests. For point-sampled two-dimensional rasterization, MSAA computations were offloaded to idle CPU cores in a shared memory architecture. The contributions to ray tracing involve a flexible BVH traversal algorithm that runs without a stack, and an efficient algorithm for traversing large streams of rays against a BVH while making use of wide SIMD.

In the future, it will be interesting to see if analytical solutions will get more traction. A limitation of analytical solutions is the difficulty to generalize the rendering algorithm to incorporate more dimensions, without reworking the entire algorithm. Algorithms based on stochastic point-sampling can often use more samples that are distributed in a higher-dimensional space. It might be interesting to investigate hybrid algorithms that combine stochastic sampling with analytical samples. For analytic solutions, level of detail becomes very important. If objects are too finely tessellated, an analytical solution may process a lot of visibility information in a single image pixel. Stochastic methods have the advantage that they can easily discard triangles that lie "between samples". To this end, compression schemes that reduce the data have been proposed [16, 36], and it will be interesting to see what happens in this space.

For ray tracing, ways to generalize ray traversal to wider SIMD is certainly interesting, and I believe that our dynamic ray stream traversal algorithm is a step in that direction. Shading efficiency is still an open problem in ray tracing, and shading entire streams of rays using SIMD execution would be an interesting avenue for future work. If ray streams are to be adopted in production, scheduling of streams must be transparent to the shading system. It would be very interesting to explore systems where shaders can send multiple rays recursively, and even wait for the result of ray queries to perform adaptive sampling, while still batching up rays in streams behind the scenes. In shared memory architectures, ray streams could be offloaded to the GPU to improve performance. Since shared memory architectures have a shared memory subsystem, it may be beneficial to reduce its strain by using a stackless traversal algorithm, thus avoiding the storage and bandwidth associated with storing a stack.

Bibliography

[1] AKELEY, K. Reality Engine Graphics. In *Proceedings of ACM SIGGRAPH* (1993), pp. 109–116.

[2] AKENINE-MÖLLER, T., AND AILA, T. Conservative and Tiled Rasterization Using a Modified Triangle Set-Up. *Journal of Graphics Tools 10*, 3 (2005), 1–8.

[3] AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. *Real-Time Rendering*, 3^{rd} ed. A. K. Peters Ltd., 2008.

[4] AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware* (2007), pp. 7–16.

[5] ANDERSSON, J. Five Major Challenges in Real-Time Rendering. In *Beyond Programmable Shading course* (2012), SIGGRAPH.

[6] CATMULL, E. A Hidden-Surface Algorithm with Anti-Aliasing. In *Computer Graphics (Proceedings of SIGGRAPH 78)* (1978), pp. 6–11.

[7] CATMULL, E. E. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, The University of Utah, 1974.

[8] CLARBERG, P., TOTH, R., HASSELGREN, J., NILSSON, J., AND AKENINE-MÖLLER, T. AMFS: Adaptive Multi-frequency Shading for Future Graphics Processors. *ACM Transactions on Graphics 33*, 4 (2014), 141:1–141:12.

[9] COOK, R. L., CARPENTER, L., AND CATMULL, E. The Reyes Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 87) 21*, 4 (1987), 95–102.

[10] ERICSON, C. *Real-Time Collision Detection*. Morgan Kaufmann Publishers Inc., 2004.

[11] ERNST, M., AND GREINER, G. Multi Bounding Volume Hierarchies. In *IEEE Interactive Ray Tracing* (2008), pp. 35–40.

[12] FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. Reducing Shading on GPUs Using Quad-Fragment Merging. *ACM Transactions on Graphics 29* (2010), 67:1–67:8.

[13] FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. Data-parallel Rasterization of Micropolygons with Defocus and Motion Blur. In *High Performance Graphics* (2009), pp. 59–68.

[14] GEORGIEV, I., KŘIVÁNEK, J., DAVIDOVIČ, T., AND SLUSALLEK, P. Light Transport Simulation with Vertex Connection and Merging. *ACM Transactions on Graphics 31*, 6 (2012), 192:1–192:10.

[15] GOLDSMITH, J., AND SALMON, J. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics & Applications 7*, 5 (May 1987), 14–20.

[16] GRIBEL, C. J., DOGGETT, M., AND AKENINE-MÖLLER, T. Analytical Motion Blur Rasterization with Compression. In *High-Performance Graphics* (2010), pp. 163–172.

[17] GRIBEL, C. J., MUNKBERG, J., HASSELGREN, J., AND AKENINE-MÖLLER, T. Theory and Analysis of Higher-order Motion Blur Rasterization. In *High-Performance Graphics Conference* (2013), pp. 7–15.

[18] HACHISUKA, T., OGAKI, S., AND JENSEN, H. W. Progressive Photon Mapping. *ACM Transactions on Graphics 27*, 5 (2008), 130:1–130:8.

[19] HACHISUKA, T., PANTALEONI, J., AND JENSEN, H. W. A Path Space Extension for Robust Light Transport Simulation. *ACM Transactions on Graphics 31*, 6 (2012), 191:1–191:10.

[20] HAMMARLUND, P., MARTINEZ, A., BAJWA, A., HILL, D., HALLNOR, E., JIANG, H., DIXON, M., DERR, M., HUNSAKER, M., KUMAR, R., OSBORNE, R., RAJWAR, R., SINGHAL, R., D'SA, R., CHAPPELL, R., KAUSHIK, S., CHENNUPATY, S., JOURDAN, S., GUNTHER, S., PIAZZA, T., AND BURTON, T. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro 34*, 2 (2014), 6–20.

[21] HENNESSEY, J. L., AND PATTERSSON, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[22] IZE, T., AND HANSEN, C. RTSAH Traversal Order for Occlusion Rays. *Computer Graphics Forum 30*, 2 (2011), 297–305.

[23] JENSEN, H. W. *Realistic Image Synthesis using Photon Mapping*. AK Peters, 2001.

[24] JONES, T. R., AND PERRY, R. N. Antialiasing with Line Samples. In *Eurographics Workshop on Rendering* (2000), pp. 197–205.

[25] KAJIYA, J. T. The Rendering Equation. *Proceedings of ACM SIGGRAPH 20* (August 1986), 143–150.

[26] LAFORTUNE, E. P., AND WILLEMS, Y. D. Bi-Directional Path Tracing. In *Proceedings of Computational Graphics and Visualization Techniques '93* (1993), pp. 145–153.

[27] LEE HOWES AND AAFTAB MUNSHI. *The OpenCL Specification 2.0*, October 2014.

[28] MACDONALD, D. J., AND BOOTH, K. S. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer 6*, 3 (May 1990), 153–166.

[29] MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proceedings of ACM SIGGRAPH* (2003), pp. 896–907.

[30] MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32.

[31] NVIDIA CORPORATION. *NVIDIA CUDA C Programming Guide*, August 2014.

[32] PINEDA, J. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of ACM SIGGRAPH* (1988), pp. 17–20.

[33] RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics 30*, 3 (2011), 17:1–17:17.

[34] STRASSER, W. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. PhD thesis, TU Berlin, 1974.

[35] SUN, X., ZHOU, K., GUO, J., XIE, G., PAN, J., WANG, W., AND GUO, B. Line segment sampling with blue-noise properties. *ACM Transactions on Graphics 32*, 4 (2013), 127:1–127:14.

[36] TZENG, S., PATNEY, A., DAVIDSON, A., EBEIDA, M. S., MITCHELL, S. A., AND OWENS, J. D. High-quality Parallel Depth-of-field Using Line Samples. In *High-Performance Graphics* (2012), pp. 23–31.

[37] VAIDYANATHAN, K., SALVI, M., TOTH, R., FOLEY, T., AKENINE-MÖLLER, T., NILSSON, J., MUNKBERG, J., HASSELGREN, J., SUGIHARA, M., CLARBERG, P., JANCZAK, T., AND LEFOHN, A. Coarse Pixel Shading. In *High-Performance Graphics Conference* (2014), pp. 9–18.

[38] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612.

[39] WILLIAMS, L. Pyramidal Parametrics. *SIGGRAPH 17*, 3 (1983), 1–11.

Paper I

High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility

Carl Johan Gribel Rasmus Barringer Tomas Akenine-Möller

Lund University

ABSTRACT

We present a novel visibility algorithm for rendering motion blur with per-pixel anti-aliasing. Our algorithm uses a number of line samples over a rectangular group of pixels, and together with the time dimension, a two-dimensional spatio-temporal visibility problem needs to be solved per line sample. In a coarse culling step, our algorithm first uses a bounding volume hierarchy to rapidly remove geometry that does not overlap with the current line sample. For the remaining triangles, we approximate each triangle's depth function, along the line and along the time dimension, with a number of patch triangles. We resolve for the final color using an analytical visibility algorithm with depth sorting, simple occlusion culling, and clipping. Shading is decoupled from visibility, and we use a shading cache for efficient reuse of shaded values. In our results, we show practically noise-free renderings of motion blur with high-quality spatial anti-aliasing and with competitive rendering times. We also demonstrate that our algorithm, with some adjustments, can be used to accurately compute motion blurred ambient occlusion.

ACM Transactions on Graphics, 30(4):54:1–54:12, 2011.

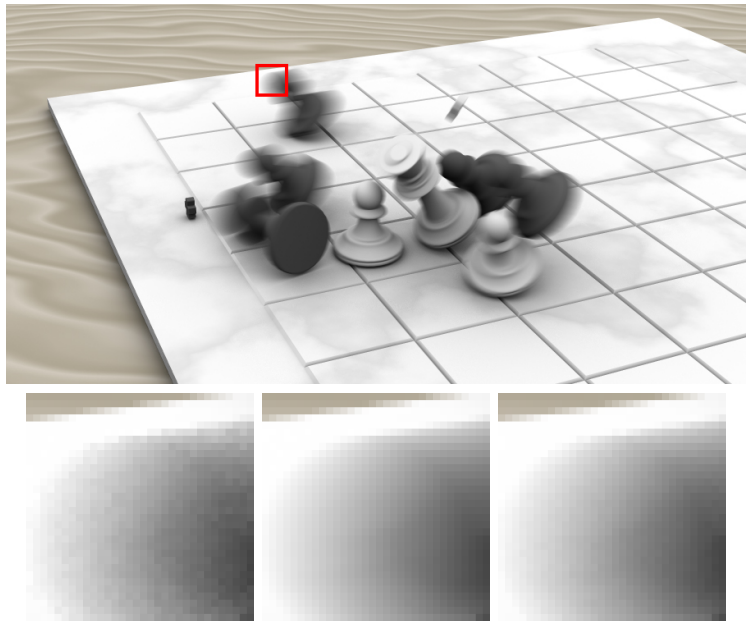


Figure 1: *A chess scene with motion blur rendered with stochastic rasterization with 49 point samples, our semi-analytical visibility algorithm in the temporal domain with four line samples in the spatial domain, and finally with stochastic rasterization with 256 point samples. Our work focuses on spatio-temporal visibility, and for 49 samples it takes* 3.8 *seconds to compute visibility and simple shading (ambient occlusion not included) at* 1024×768 *pixels. With these settings, our algorithm computes the middle image in* 3.6 *seconds. Note that the image with 49 samples is rather noisy, and even with 256 samples, there is still some noise, while the motion in our image is essentially free of noise. Furthermore, the quality of the spatial anti-aliasing (look at the static edge at the top) in our image closely matches that of 256 point samples.*

1 Introduction

Visibility computations is a fundamental core research topic in computer graphics, and it has been active and vivid for more than 45 years. Algorithms for visibility play a central role in essentially any type of rendering including, for example, rasterization, ray tracing, two-dimensional graphics, font rendering, shadow generation, global illumination, and volume visualization.

During the 1970's and 1980's, research on analytical visibility for spatial anti-aliasing [5, 30] and motion blur [20, 6, 13] was rather popular. However, after

Cook et al.'s stochastic point sampling approaches were presented [8, 7], such techniques pretty much fell into oblivion. Instead, visibility was either solved using a depth buffer [4] or using ray tracing [32], and most often with some type of point sampling.

An interesting observation by the computer science community is that the gap between available compute power and memory bandwidth is large, and continues to grow rapidly [18, 25]. In addition, the power consumption by a memory access and a floating-point operation differs by at least an order of a magnitude [9]. Hence, common advice today is to refactor an algorithm so that it instead uses more computations and fewer memory accesses. With this development of computer architecture, one logical consequence is that it makes more sense to (again) explore analytical visibility computations, which are computationally more expensive than point sampling techniques.

To that end, we present a new visibility engine which is loosely based on previous work on analytical visibility for spatial anti-aliasing [5] and on analytical motion blur with spatial point sampling [15]. Our goal is to generate high-quality images with near-perfect anti-aliasing in both the spatial domain and in the temporal domain. To reduce the dimensionality of the problem, we use line samples [19] in screen space, and solve for analytical visibility along such lines and over time. We present a novel visibility engine for this, and show that our algorithm can rapidly generate practically noise-free images on many-core computers. We also show that a variant of our technique can render ambient occlusion with motion blur.

2 Previous Work

There is a wealth of research in the visibility field, and in this section, we review only the research that is most relevant to our work. This means that we avoid discussing approaches based on point sampling visibility, such as multi-dimensional adaptive sampling [16], for example.

Cook et al. [7] presented the REYES rendering system using rasterization of motion blur and depth of field with stochastic point sampling [8]. Lately, much effort has been spent on stochastic rasterization of (micro-)polygons for motion blur and depth of field [1, 10, 23]. Ragan-Kelley et al. present a hardware architecture for rasterizing motion blur and depth of field [27]. By decoupling shading from visibility, they essentially extend the concept of multi-sample anti-aliasing to motion blur and depth of field. While REYES' target was offline rendering, this later line of research is targeting interactive or even real-time graphics. In our current work, point sampling is only used for shading.

Catmull [5] presented a visibility algorithm that processes the polygons from top to bottom, and from left to right in order to exploit coherence of the scene. A per-pixel clipping algorithm is applied where each edge clips the existing polygons into two groups recursively in order to compute the exact polygon coverage, with excellent spatial anti-aliasing as a result. The algorithm by Weiler and Atherton [30]

is rather similar, but also describes how to render shadows and transclucent geometry. Korein and Badler [20] presented an analytical visibility algorithm for motion blur, where each spatial point sample computed analytical coverage of disks over time. After all geometry had been processed, a hidden surface removal algorithm was applied to resolve for final sample color. The details of how this technique can be extended to linearly moving triangles are given by Gribel et al. [15]. Korein and Badler were also the first to present accumulation buffering, which can be used for all sorts of point sampling on a per-frame basis. Accumulation buffering has been used for spatial anti-aliasing, depth of field, and motion blur [17, 31].

Sung et al. [28] decouple shading from visibility on a per-object basis, somewhat similar to Burns et al. [3], and use point sampling for spatial antialiasing, with analytical visibility over time per point sample [20]. In our work, we also use decoupled shading from visibility, and fall back on point sampling of shading, with efficient reuse using a cache [27, 3].

Jones and Perry [19] presented a "near-analytic" screen-space anti-aliasing technique that uses line samples to reduce dimensionality of the problem. All polygons in the scene are projected onto these lines, and visibility resolved along the line with linear depth per polygon. They use horizontal and vertical line samples through the center of the pixels, and blend the results of these according to the edges inside the pixel. A similar line of work was presented by Gribel et al. [15], where point sampling was used in screen space, and "line samples" were used to analytically compute motion blur. They also presented a lossy compression algorithm in order to handle a large number of triangles per pixel. In our work, we combine the work of Jones and Perry, Gribel et al., and Catmull to obtain an algorithm for high-quality spatial anti-aliasing with motion blur.

Recently, Manson and Schaefer introduced wavelet rasterization [22], which can *analytically* rasterize polygons and Bézier curves in two dimensions, and also three-dimensional meshes into voxel grids. They rasterize the primitives into a hierarchical Haar wavelet tree representation and show that it is robust to degenerate input. This is really interesting work that could potentially be applied to motion blur as well. Finally, we refer to the survey by Sutherland et al. [29] for an overview of ten visibility algorithms pre-dating the depth buffer [4], and to the overview by Sung et al. [28] for spatio-temporal anti-aliasing.

3 Algorithm Overview

In this section, we present an overview of our algorithm for generating images with high-quality spatio-temporal anti-aliasing. In addition, we will first explain some key concepts around our sampling strategy using analytical visibility computations. We assume that the geometry consists of triangles, and that each triangle vertex can move linearly in world space over time, $t \in [0, 1]$.

Instead of using multiple point samples over a pixel for spatial anti-aliasing, we choose to use a set of line samples [19]. We define a *line sample* as an arbitrary

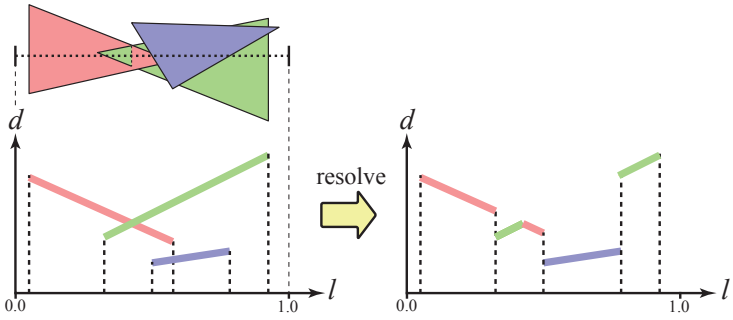


Figure 2: *At the top, a configuration of three triangles (in screen space) intersecting a line sample (dotted line) is shown, and just below, the depth functions, d, for the three triangles on the line sample are illustrated. To the right, we have resolved for the closest depth segment over the parameter, l.*

straight line over one or more pixels, and we will compute visibility analytically over such lines. Over each line sample, Jones and Perry determine which triangles overlap with the line sample, and compute the depth, d, at the end points of each visibile segment. The depth, $d = z/w$, is linear in screen space [2], which means that the depth at the end points are sufficient to store. See Figure 2 for an example.

Geometrically, a line sample can be thought of as a three-dimensional triangle with one vertex at the camera position, and going through the end points of the line sample in world space, and extending infinitely. Occasionally, we will refer to the plane of this triangle as the *line sample plane*. A line sample is illustrated as a blue line to the left in Figure 3. We parameterize along each line using a parameter, $l \in [0,1]$. A core difference compared to other spatio-temporal visibility algorithms is that we resolve visibility in the lt-space, where $t \in [0,1]$ represents the time dimension, and $l \in [0,1]$ is the parameter along a line sample. For a static triangle, the intersection between the triangle and the line sample plane will be a single line,[1] which is the same for all values of t. This is illustrated in the lt-space to the left in Figure 3. When a triangle starts to move, however, the moving triangle's line of intersection will change over time, and will trace out a different region in the lt-space. An example is shown to the right in Figure 3.

The motivation for using line samples is manifold. First, a line sample can be thought of as infinitely many point samples on the line, and therefore has potential for fast convergence. Second, as mentioned in the introduction, trading computations for bandwidth is a general advice on today's computing architectures. Finally, using line samples instead of performing a full analytical resolve in the xyt-space reduces the problem from three to two dimensions which makes it more tractable. In practice, we use sampling patterns so that 2–4 line samples will

[1] Except when the triangle lies in the plane, in which case we will cull it from further processing.

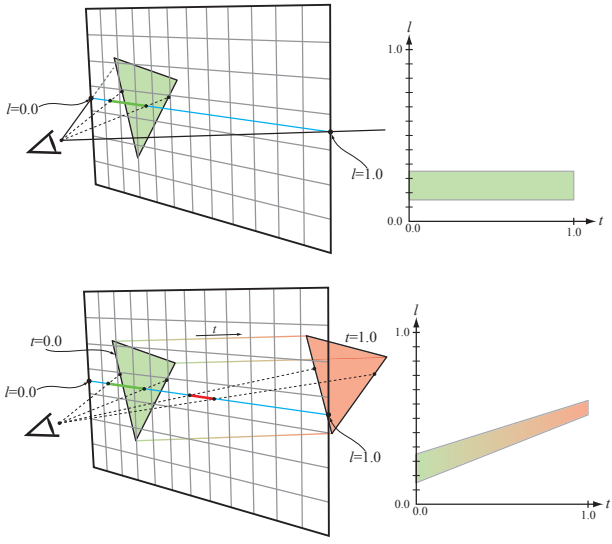


Figure 3: *Top: a static triangle rendered over a* 12×7 *pixel grid. We show a single line sample (blue line) extending through a row of pixels. The line sample plane is defined by the camera position and the line sample. The lt-space shows the triangle coverage over the line sample and over time, which in this case is a rectangle covering the entire time interval. Bottom: a triangle being translated over* $t \in [0,1]$*. The triangle is visualized as green at* $t = 0$*, red at* $t = 1$*, and each vertex moves linearly in between. As can be seen, when the triangle moves, it traces out a different region in the lt-space. Each point,* (l,t)*, will also have a depth, d, but that is not visualized here—see Section 4.*

intersect each pixel, and these patterns are discussed in Section 7.

On a high level, our entire algorithm works as follows. First, a bounding volume hierarchy (BHV) is built over all geometry in the scene. A small rectangular region, called a *tile*, of the entire image is rendered at a time. This allows for parallel execution on many threads since a core can render to a tile independently of others. For each tile, the BVH is traversed so that only geometry that overlaps with the tile is processed. In the next step, each line sample inside the tile is processed one at a time. Every triangle that intersects a line sample is represented as a *depth patch* in ltd-space, where each point, (l,t), has a depth, d. These depth patches can form complex surfaces, and in Section 4 and in Appendix A, we present how we construct these patches and approximate them with, what we call, *patch triangles* in the ltd-space. We call our algorithm *semi-analytical*, since most computations are analytical, except for the approximation of the depth patches.

The patch triangles approximating a depth patch are sent to our visibility engine (Section 5), which resolves for depth visibility over the lt-space. This part of

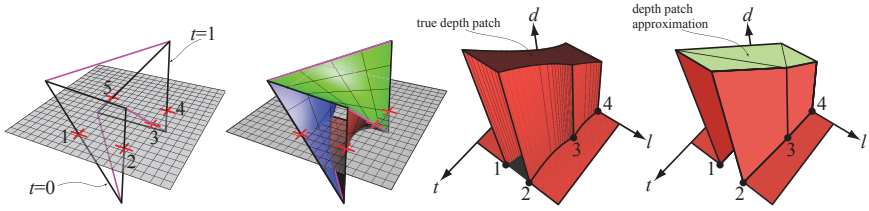


Figure 4: *Left: illustration of a moving triangle intersecting the line sample plane (gray). The triangles at time $t = 0$ and $t = 1$ have black outlines, and the motion vectors are purple. The feature points are marked with red crosses, and are all the intersections between the line sample plane and the triangle edges and motion vectors. Middle-left: visualization of the bilinear patches that are traced out over time with linear motion per vertex. Middle-right: the true depth patch in ltd-space. We generated this image using point sampling. Right: for this depth patch, a possible approximation consists of the three green triangles, called patch triangles.*

our algorithm was inspired by Catmull's analytical screen-space anti-aliasing algorithm [5]. When all triangles have been processed, final visibility is resolved, and for each pixel, the contribution of the line samples overlapping the pixel is accumulated to that pixel's color. In Section 6, we also show that a variation of our algorithm can be used to compute motion blurred ambient occlusion using "line samples" on the hemi-sphere.

4 Depth Patches

As illustrated in Figure 3, each moving triangle intersecting a line sample will give rise to some region in lt-space. In addition, each point, (l,t), will also have a depth, d. We call these surfaces in ltd-space *depth patches*. Figure 4 shows that these depth patches can form rather complex surfaces, and it is quite obvious that there is little hope in finding an analytical representation of these that also is fast to process. Therefore, our approach is to approximate the exact depth patches with a number of triangles, here called *patch triangles*, in ltd-space. We use the following notation for a triangle in world space. A linearly moving vertex is denoted $\mathbf{p}(t) = (1-t)\mathbf{q} + t\mathbf{r}$, and a triangle is simply three vertices, $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$. Hence, the triangle at $t = 0$ is $\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, and the triangle at $t = 1$ is $\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$. The term *motion vector* is used for $\mathbf{r}_i - \mathbf{q}_i$, and *triangle edge* for any triangle edge at $t = 0$ or $t = 1$. Next, we describe how our patch triangles are constructed.

Note that the entire surface of a moving triangle consists of the triangles at $t = 0$ and $t = 1$, and three bilinear patches, each generated by a moving edge. The intersection between such a moving triangle and a line sample plane will trace out one or more connected regions in lt-space. The boundary of such a region is called a *patch outline*, and consists of connected straight lines and curves. This is il-

lustrated in Figure 4. The straight lines originate from the intersections between the line sample plane and the triangles at $t = 0$ & $t = 1$, and the curved segments will be generated by the intersections between moving edges and the plane. Next, we argue that the points in lt-space, where lines and/or curved segments are connected, are easily generated. We call them *feature points*.

First, we observe that the triangles at $t = 0$ and $t = 1$ will intersect the line sample plane in lines in the lt-space, and hence the intersections between the plane and the triangle edges are feature points. Second, we note that a moving edge traces out a bilinear patch, which is a ruled surface. This means that the intersection curve(s)[2] between a bilinear patch and a plane will be curves that start and end in the intersection points between the plane and the four edges of the bilinear patch. Hence, these intersection points are also feature points. Therefore, we define the feature points as all the intersections between the line sample plane and the triangle edges & the motion vectors. The triangles at $t = 0$ and $t = 1$ can *each* give rise to at most two feature points. In addition, each motion vector can give rise to one feature point. In total, this sums to at most $2+2+3 = 7$ feature points per moving triangle.

These feature points are marked with red crosses to the left in Figure 4, and at each feature point, it is straightforward to compute its depth, d. Depending on the configuration of these feature points, they need to be connected in a certain order. This is an important implementation detail, and so in Appendix A, we describe how the feature points are used to tessellate the true depth patch into a number of *patch triangles*. In that appendix, we also describe how the tessellation can be adapted to rapid change in the depth patch function. For the next section, we assume that our visibility engine receives a stream of patch triangles approximating the true depth patch in ltd-space for each moving triangle.

5 Visibility Engine

The purpose of our visibility engine is to analytically resolve visibility for a spatial line sample and ultimately compute the color contribution from that line sample to each pixel. As mentioned in Section 3, we process a rectangular block of pixels, called a *tile*, at a time. Our geometry is represented in a three-dimensional bounding volume hierarchy (BVH), where each leaf node may contain one or more moving triangles. We cull the BVH against the frustum spanned by the tile, and we further cull the remaining geometry against the line sample plane.

At this point, we have a set of moving triangles that overlap with the line sample, and we would like to find all *visible* patch triangles (Section 4) in both the spatial and in the temporal domain, i.e., in the lt-space. In order to support near-z clipping, all triangles are tested against a line sample plane in view-space. The resulting patch triangles are then individually clipped against near-z, projected into

[2]There can be one or two such curves—see Appendix A.

viewport-space, and mapped along the sample line into lt-space. The lt-space is diced up into $m \times n$ uniform grid cells, where m is the number of pixels that the line sample overlaps with, and n is a user-defined constant specifying the number of subdivisions in the t-dimension. The grid cells are referred to as lt-cells, or simply just cells. Our visibility engine operates completely in the lt-space, and can be divided into three stages, namely, *binning*, *depth sorting*, and *pixel integration*. Next, these three stages are described in more detail.

5.1 Binning

The binning stage processes one patch triangle at a time, and finds all patch triangles that overlap with each lt-cell. This is done by conservatively rasterizing the patch triangle to the lt-cells. For all patch triangles overlapping a cell, a pointer to that patch triangle is stored in a list of that cell. In addition, we perform a simple variant of occlusion culling [14] in this stage. If a patch triangle is found to completely cover a cell, the maximum depth, $Z_{\max}^{\text{cell}}$, of the patch triangle over the cell is computed. That cell is then listed as "fully covered" with $Z_{\max}^{\text{cell}}$. All subsequent patch triangles that overlap with a fully covered cell can be occlusion culled (and not added to the cell list) if they are farther away than that cell's $Z_{\max}^{\text{cell}}$. In order to occlusion cull even when triangles are not rendered strictly front to back we also store $Z_{\min}^{\text{cell}}$. If a newly added triangle covers the entire cell and its $Z_{\max}^{\text{tri}}$ is less than $Z_{\min}^{\text{cell}}$, the entire cell can be cleared. After all patch triangles have been processed, all potentially visible patch triangles that overlap with a cell in the lt-space are known. It is the task of the following two stages to resolve for final visibility.

5.2 Depth Sorting

The goal of depth sorting is to deliver a list, for each lt-cell, of non-intersecting polygons sorted according to ascending depth. In order to ensure depth order inside each lt-cell, a local BSP-tree [11] is created from the patch triangles in each cell's list. The patch triangles are added to the BSP-tree in turn, which creates leaf nodes with their splitting plane taken from the patch triangle plane. Subsequent patch triangles may be split against these planes during insertion into the BSP-tree. Once the BSP-tree has been built, depth sorting is simply a matter of traversing the BSP-tree. Note that on average, our BSP-trees are small since they only cover one pixel's extent on a line sample, and a certain span in time. For our test scenes, we usually have less than 100 triangles per lt-cell, and hence creation and traversal is relatively fast.

5.3 Pixel Integration

The goal of the pixel integration is to calculate the color contribution of a sample line to each pixel it overlaps. Once the color for all lt-cells covering a pixel is known, the results can be weighted together to give the final color of the pixel. In

order to integrate only the visible part of each polygon, a hidden surface algorithm is used to eliminate occluded geometry in each lt-cell. By traversing the BSP-tree per lt-cell, we obtain a strictly depth sorted list of polygons. This means that once a region in an lt-cell has been found to be covered by a polygon, no other polygon can cover that region.

The hidden surface algorithm is similar to "the pixel integrator" proposed by Catmull [5]. This algorithm inserts all depth sorted polygons into a tree in order of front to back. Each added polygon is split against the edges of the polygons already in the tree. The part to the left of the edge is added as a child to that edge in the tree. The remaining part is split against the next edge. This is illustrated in Figure 5.

After hidden surface removal, calculating the color of the lt-cell is a simple matter of summing the color of each visible (convex) polygon weighted by its visible area, possibly weighted by a filter kernel. The color of each polygon is determined by invoking the pixel shader. We currently shade each convex polygon at its barycenter, which is a natural place to put it, and this also generated high-quality images. Hence, n is connected to both the number of cells in time as well as the shading frequency (since shading is performed at least once for each patch triangle in each cell) in our current implementation. Texture differentials are calculated analytically using the triangle at $t = 0$.

Discussion As mentioned, our visiblity engine was inspired by Catmull's early work [5] that operates in the spatial domain (xy) without motion blur. However, for our work, there are a number of subtle but important differences. Our visibility engine and resolve procedure are working entirely in lt-space, which enables motion blur to be handled. In the binning stage, we process one patch triangle at a time until all patch triangles have been completely binned, which makes our engine similar to feed-forward rasterization. This is in contrast to Catmulls' scanline order processing which requires sorting on l, and handling of every patch triangle covering a scanline before proceeding to the next. In addition, we have added a simple form of occlusion culling for better efficiency. All images in Catmull's paper [5] appear to consist of 2D layers composited with anti-aliasing, but we need general sorting, and therefore use a BSP-tree instead of assuming already sorted geometry.

6 Ambient Occlusion

In this section, we will show that, with some small modifications, our depth patches (Section 4) and our visibility engine (Section 5) also can be used to compute *motion blurred* ambient occlusion (AO). To the best of our knowlege, this is a topic that has received very little attention in graphics research. We start with a brief introduction to ambient occlusion, and then describe how our algorithms can be used for static and motion blurred ambient occlusion in subsequent subsections.

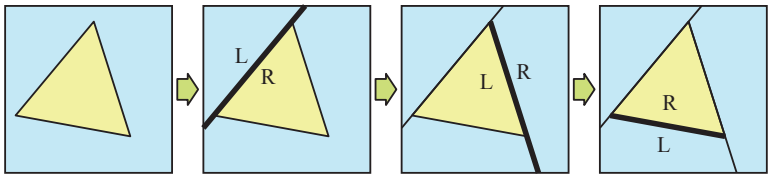


Figure 5: *A patch triangle shown inside an lt-cell. In the pixel integrator, the first triangle edge splits the cell into a left (L) region and a right (R) region. In this case, the next edge splits the previous right region into two new regions, and so on. This creates a tree, and each new triangle is clipped against the edges in the tree, and possibly inserted into the tree.*

Zhukov et al. [33] define the amount of ambient occlusion, o, as:

$$o(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} \rho\left(d(\mathbf{p},\omega)\right)(\omega\cdot\mathbf{n})d\omega, \tag{1}$$

where $\mathbf{p}$ is the point where AO is computed and $\mathbf{n}$ is that point's normal. The integral is over the hemisphere, Ω, with its "north pole" in the direction of $\mathbf{n}$. The distance function, d, returns the distance to the closest occluder in direction ω, and ρ is a distance fall-off function, which has a value between 0 (fullly occluded) and 1 (not occluded). Intuitively, the equation computes how much of a white hemisphere that a diffuse receiver can "see." For high-quality renderings, one may need a thousand rays to compute AO using point sampling [21], and for motion blurred point-sampled AO, we expect that even more rays may be needed.

6.1 Static Ambient Occlusion

Instead of integrating over the hemisphere using point samples, we extend the concept of line samples to the hemisphere. Here, we define a line sample as the intersection between the hemisphere, and a plane going through the center of the hemisphere. An example is the orange plane shown to the left in Figure 6. In the following, we will first concentrate on a single line sample. However, note that a number of line samples will be needed to accurately sample AO over the hemisphere, and we will return to line sample distributions later in this section. For all triangles above or overlapping the hemisphere base plane (going through the "equator"), we compute the intersection (if any) between the triangle and the line sample plane, and project the intersection onto the line sample's hemicircle. The next step is to project that down to the base plane as shown to the right in Figure 6, and there we illustrate our parameterization, l, of the line sample on the hemisphere. Note that this implements the Nusselt analog, which means that the cosine factor of Equation 1 is included by design per line sample.

Similar to line sampling in screen space, we also use linear depth segments and resolve for visibility with respect to depth exactly as shown in Figure 2, and de-

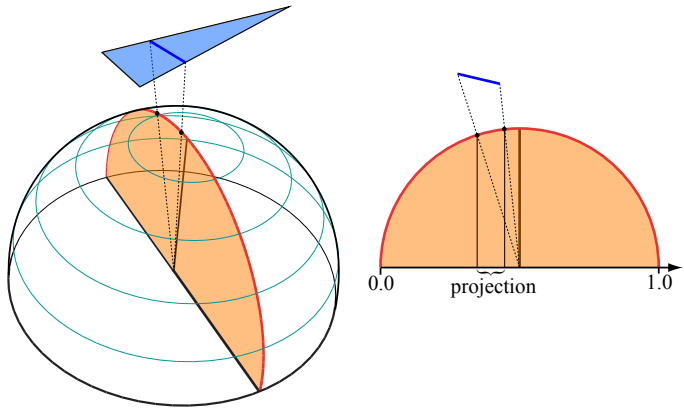


Figure 6: *Our approach for ambient occlusion is to use a number of line samples over the hemisphere. In this illustration, only a single line sample is shown, but in practice, several will be used to cover the hemisphere. Left: a line sample in this case corresponds to a hemi-circle (red), and all triangles overlapping with the plane (orange) through that hemicircle are projected onto the hemicircle. Right: illustration of the projection of an intersection between a triangle and the line sample plane in two dimensions. As can be seen, the intersection is first projected onto the hemicircle, and then down to the black line, which is parameterized by l.*

scribed by Gribel et al. [15]. However, note that depth is not linear over l in this case. If the goal is to only compute whether the hemisphere is occluded or not, then it does not matter that depth is not linear, since we are only interested in occlusion and not depth. However, if the falloff function, ρ, in Equation 1, needs to be taken into account, depth matters. In this case, long projections on l must be diced up into several shorter linear depth segments, which makes for a better approximation.

The remaining part at this point is line sample distributions over the hemisphere. As far as we know, this is a rather unexplored topic in sampling. To the left in Figure 7, a simple sampling pattern is shown, where all eight line samples are passing through the "north pole" of the hemisphere. In general, we strive after uniform distributions in xy-space, and it is quite clear that this distribution is non-uniform. To counteract that, we instead redistribute the projection on l before they are inserted. In this case, this amounts to a quadratic remapping function, similar to that used for depth of field remapping [1]. Without remapping, we also measure line density as suggested by Grabli et al. [12], where we used a measurement radius of 0.5. Such images are also shown in Figure 7. As can be seen to the right in the same figure, a more uniform distribution can be obtained if the line sample planes also are allowed to tilt. This makes the mapping parameterization a bit more complex, but we expect that it will be worth the effort.

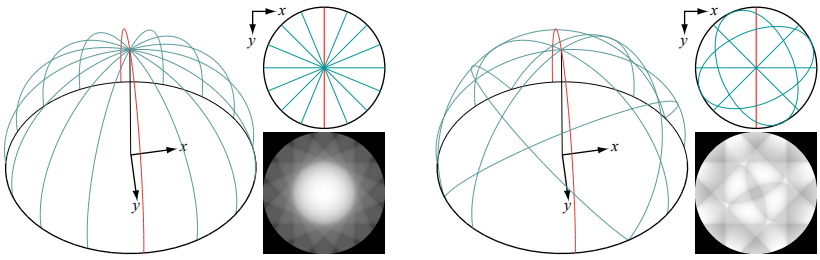


Figure 7: *Left: a line distribution with eight rotated planes going through the north pole of the hemisphere. The lines projected to xy, and their line density image [Grabli et al. 2004] are also shown. Since this distribution is rather non-uniform, we remap the coordinates along l in order to obtain a uniform distribution. Right: a more uniform distribution of lines in the xy circle of the hemisphere. As can be seen, the line density image is more uniform for this distribution.*

6.2 Motion Blurred Ambient Occlusion

Our extension to motion blurred ambient occlusion (AO) is rather straightforward. Instead of just processing visibility along the l-axis (Section 6.1), we now compute visibility in lt-space as described in Section 4 and 5. For static receivers of AO, this technique works as expected. A moving sphere over a plane will cast a motion blurred AO shadow on the static plane receiver, for example.

For dynamic receivers, this situation is more complex, because both the receiving sample position on a surface, and its normal may change as a function of time. We avoid this complexity by considering all motion relative to a local coordinate system at the sample point. The occluders are transformed into this local coordinate system at both $t = 0$ and $t = 1$. The relative motion of each vertex is approximated as linear with respect to time, which we believe is reasonable because that is exactly how vertex motion is described in our system. Note also that we clip the patch triangles against $z = 0$ so that parts of patch triangles "below" the hemisphere are not processed.

As described in Appendix B, we use a shading cache in order to efficiently reuse shaded values over the time dimension. We note that shading caches [27, 3] are two-dimensional over some parameterization of the rendered surfaces, and for motion blur, the time, t, is collapsed and always computed at $t = 0$, for example. This has direct consequences for shadows and AO when they are point sampled in time. Shadowing at a moving surface point will simply be evaluated at $t = 0$, and if the surface point later in time changes from being in shadow to being lit, that will not be accounted for. This can easily give rise to images with unnatural looking shadows. This is a limitation of shading caches, and it may be an interesting avenue for future work to extend shading caches to handle motion blurred shadows and AO better.

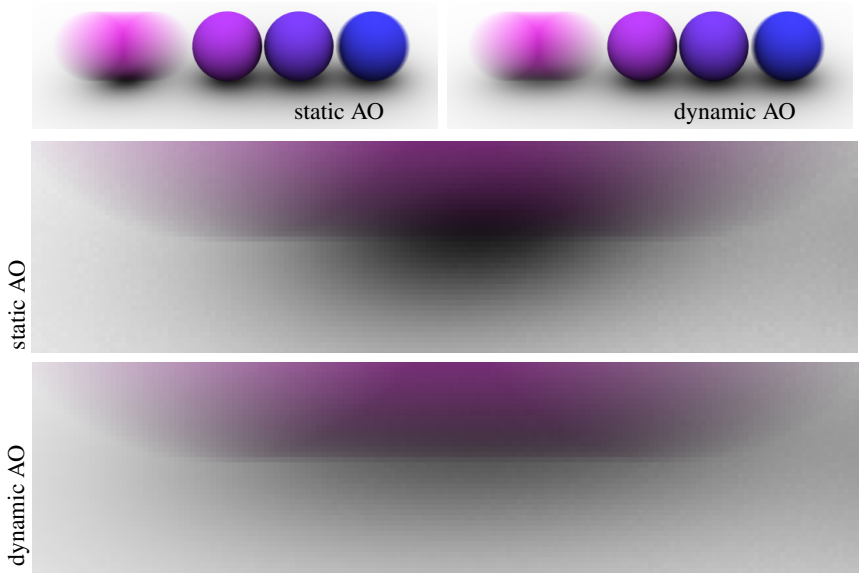


Figure 8: *Static versus motion blurred (dynamic) ambient occlusion (AO) with 16 line samples for AO. For the static case, AO was computed at $t = 0.5$. Dynamic AO renders more plausible images as seen in the close-ups.*

Interestingly, our solution will not compute AO at $t = 0$ and reuse this value over all samples. As described above, our algorithm will compute the AO in lt-space, and it is the average AO over time that will be computed and put into the shading cache. For static receivers, this generates a correct image. However, for moving receivers, this is not strictly correct, but we believe the average over time is a better solution than computing AO at a single instant of time and reuse. In Figure 8, we show the difference between static AO and motion blurred AO.

7 Implementation

We have implemented all our algorithms in a custom renderer in C++, and have the pixel processing part of the algorithm threaded in order to exploit more than one CPU core. Our algorithm is compared against stochastic rasterization with efficient backface and view frustum culling. In addition, we have an early-Z depth test and a simple form of occlusion culling [14] with one $Z_{\max}$-value per 8×8 pixels. For stochastic rasterization, we use multi-jittered sampling points. In the following, we will discuss line sampling patterns for screen space sampling, which is not a well explored topic in graphics.

Some obvious line sampling patterns to test include using one horizontal and one

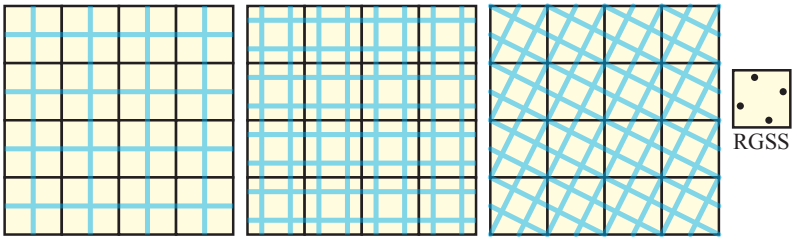


Figure 9: *Line sample patterns for* 4×4 *pixels. Left: one horizontal and one vertical (1H1V) line sample (blue) is going through each pixel. Middle: two horizontal and two vertical (2H2V) line samples. Right: a rotated grid of line samples (RGLS), inspired by the point sampling pattern called RGSS.*

vertical line sample per pixel. This is what Jones and Perry used in their inspirational research [19]. Instead of creating two unique lines per pixel, we extend the lines so they start and end at the sides of the current tile in order to better exploit coherency. In general, this reduces the number of BVH traversals and patch triangle generations needed per rendered image. For better quality, one can increase to two horizontal and two vertical line samples per pixel. Such schemes are shown to the left in Figure 9. However, it is well-known that humans are most sensitive to jaggies on near-horizontal and near-vertical edges, and thereafter on edges which are close to 45° [24]. This clearly motivates low-cost point sampling patterns, such as rotated grid super sampling (RGSS). To that end, we experimented with a line sampling pattern that exhibit similar charateristics as RGSS, but is also designed to share long line samples across many pixels. This pattern, which we call rotated grid line sampling (RGLS), is shown to the right in Figure 9. While RGLS perform much better compared to using horizontal and vertical line samples, we leave the optimization of line sample patterns for future work. This is an important topic that we intend to revisit in later research.

8 Results

We have rendered our images using a Mac Pro 5.1 with two six-core Intel Xeon CPUs at 2.93 Ghz and 8 GB of memory. We use a decoupled shading cache as described in Appendix B for both our algorithm and for stochastic rasterization of motion blur. For one core, we shade about 10% of the requested shading values for 64 stochastic samples. This is deliberately rather high because we want high-quality shading. For 24 threads, this increases to 21% due to the fact that some shading computations are computed more than once (see Appendix B). Some details of our stochastic rasterization implementation are given in Section 7. In all our renderings, we use a tile size of 32×32 pixels. Concerning tile sizes, we have noted that our algorithm is not very sensitive to bin spread. For example, rendering

the chess scene at 512×512 resolution using 16×16 tiles requires only 30-35% more time than using 512×512 tiles, which is equivalent to sort-last. We have used three main test scenes to evaluate our algorithm. The *chess* scene in Figure 1 has 29,068 triangles and procedural texturing, *Sponza* in Figure 11 has 66,450 triangles with lighting pre-baked into textures, and *breaking armadillo* in Figure 12 has 88,120 triangles (see video). The majority of the triangles in chess and armadillo are subpixel-sized at 1024×768 pixels. In addition, we constructed two special scenes for a detailed performance analysis; the *trees* scene with 273,468 triangles and high depth complexity (see Figure 13), as well as a simple *plane* scene with varying tessellation (see Figure 14).

The line sample patterns for spatial anti-aliasing in Figure 9 are evaluated in Figure 10 for one static frame in the breaking armadillo scene. As can be seen, the RGLS scheme is superior to the other two. The main reason for this is that rotated patterns are better at combating aliasing on near-horizontal and near-vertical edges, which visually suffer the most from aliasing [24]. In general, a pattern's edge anti-aliasing effectiveness depends on the angle between the lines of the sampling pattern and the edges of the geometry to be rendered. Furthermore, in our case, there is a small rotated square in the middle of each pixel, and a triangle that falls completely within this area can disappear. However, this is usually not a problem for connected geometrical objects, but can generate artifacts for single, disconnected triangles. In terms of performance, one horizontal and one vertical (1H1V) line sample per pixel is about twice as fast as two horizontal and two vertical (2H2V), which is to be expected since there are twice the number of line samples in 2H2V. In addition, 2H2V is about 10% faster than RGLS due to the fact there are more and longer (on average) line samples per tile in RGLS compared to 2H2V.

Since motion blurred visibility is the major focus of our paper, we first report rendering times without ambient occlusion for our three test scenes, all at 1024×768 pixels with RGLS. As seen in Figure 1, we can render the chess scene in 3.6 seconds (s) with our algorithm. The number of subdivisions in time (see Section 5) was set to $n = 1$, which worked well due to rather low-frequency shading. An image with 49 samples per pixel using stochastic rasterization (SR) took 3.8s to render, and for 256 samples, 27s were needed. If we increase the amount of motion by 50%, our algorithm renders the chess scene in 4.6s, while SR uses 4.0s. However, the SR generated image contains substantially more noise (see video) in the regions with motion, while our image remains free of noise. Hence, if noise-free images are needed, it is clear that our visibility algorithm is very competitive.

The Sponza scene, in Figure 11, has camera motion. For our algorithm, we have implemented near-plane clipping, but we have not done so for stochastic rasterization (SR). Therefore, we cannot render Sponza with SR, and we note that this actually gives SR a little speed advantage in our other timings, since clipping is not done for SR. Here, we used $n = 16$ to ensure high-quality renderings. At 1024×768, this image took 20.1s to render. In Figure 12, we show the breaking armadillo scene. As can be seen, the break-even point compared to stochastic ras-

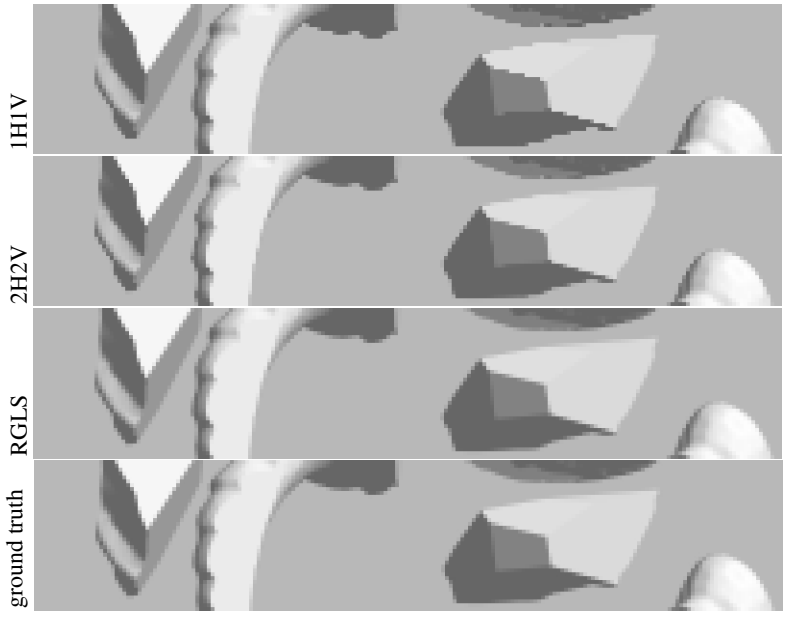


Figure 10: *Here we compare the line sampling pattern quality for a static frame of the breaking armadillo. As can be seen, rotated grid line sampling (RGLS) is substantially closer to the ground truth (256 stochastic samples) than the other two.*

terization is around 144 samples per pixel, which is much higher than for the chess scene. This is due to the armadillo having more triangles that also are located in a small volume of the scene.

In order to investigate how our algorithm behaves for high depth complexity in combination with increasing motion, we measured rendering time with respect to increasing motion for various values of n. This is shown in Figure 13. The results suggest that performance scales linearly with increasing motion, given that an optimal value of n is chosen. This implies that an adaptive splitting scheme in time is worth investigating in future work.

In Figure 14, a more detailed performance analysis of the different stages of the algorithm is shown. The plane scene is rendered using a single core with different tessellation rates as well as different values for n. Depth sorting and hidden surface removal are clearly the bottlenecks with high tessellation and motion when using a low value of n. These stages are most affected by a high number of triangles per lt-cell. As expected, increasing n greatly reduces the aforementioned stages. Patch generation time appears to be superlinear with increasing motion. The reason for this is that our BVH becomes less efficient for large amount of motion.

To analyze the approximation error introduced by our algorithm, we made a com-

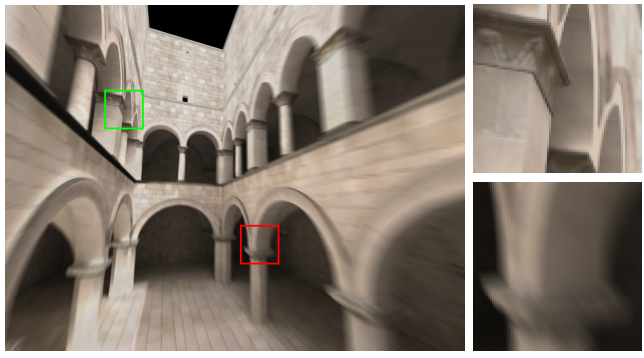


Figure 11: *The Sponza scene with camera motion, rendered with our algorithm, and with close-ups on both geometric edges (top right) and on a region with lots of motion (bottom right).*

parison to a stochastic rendering of the chess scene. The differences in depth, d, were computed and expressed as a fraction of $Z_{\text{far}} - Z_{\text{near}}$, and compiled into a histogram as shown in Figure 15. Note that our approximation also introduces errors in lt, i.e., the patch outline, and this may alter the geometrical silhouette, which means that sometimes the correct geometrical object will be missed. In such cases, the depth errors may be substantial. Nevertheless, for zero subdivisions of the patch outline (described in Appendix A.1), 99.82% of the errors fall within $< 0.1\%$ of $Z_{\text{far}} - Z_{\text{near}}$. For three subdivisions, this increases to 99.95%. This implies that the approximations are reasonable. The heat maps of the scene in Figure 15 visually shows the distribution and magnitude of the errors. More importantly, a direct comparison between our rendered images to the ground truth is included. One area with high-density errors is the pedestal bottom of the tumbling chess piece to the left. Here, depth varies significantly over time, which causes greater errors due to linearisation, but as can be seen, visual errors are still hard to detect.

With some adjustments, our visibility algorithm can also be used to compute ambient occlusion (AO) both for static and for motion blurred scenes. This is a nice side effect described in Section 6. Except for point sampling AO in time, we are not aware of any algorithms that specialize in motion blurred AO, and yet, the difference in the rendered images can be substantial, as shown in Figure 8 and 16. In all our renderings with AO, we used line sampling distributions where all line samples go through the "north pole" of the hemisphere, as shown to the left in Figure 7. As expected, doubling the number of line samples doubles the time spent on computing AO. Renderings with different number of line samples are shown in Figure 17. For the chess scene with eight line samples for AO and $n = 4$, rendering with static AO took 251 seconds, and with motion blurred AO, this increased to 2075 seconds. We have seen similar increase in rendering times for the other

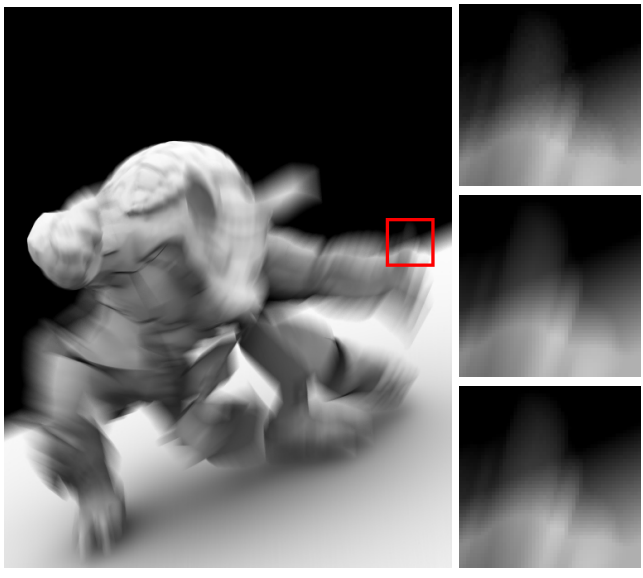


Figure 12: *Left: Armadillo with ambient occlusion with 256 stochastic samples. The images to the right are close-ups of armadillo's moving arm without ambient occlusion. Right-top: using 144 samples per pixel with stochastic rasterization rendered in 10.4s. Right-middle: close-up rendered with our analytical algorithm in 9.8s. Right-bottom: ground truth with 625 samples.*

scenes. The focus of our work has been on motion blurred visibility in screen space with spatial anti-aliasing, and hence, the evaluation of AO is not as rigorous. In future work, we will therefore compare both rendering speed and image quality of our AO algorithms (both static and motion blurred) against stochastic point sampling, and test line sampling patterns where the planes are allowed to tilt (right in Figure 7), and optimize the placement of the planes. In addition, we will also optimize for rendering speed.

9 Conclusions and Future Work

To the best of our knowledge, we have presented the *first* visibility algorithm that computes semi-analytical motion blur over spatial line samples, and we have shown that essentially noise-free images can be generated with competitive rendering times. By using line samples in the spatial domain, our visibility problem for motion blur becomes a two-dimensional problem, which makes it tractable and efficient. The approximation we have introduced is in the depth function of a moving triangle over a line sample, and despite this, we have shown that our ap-

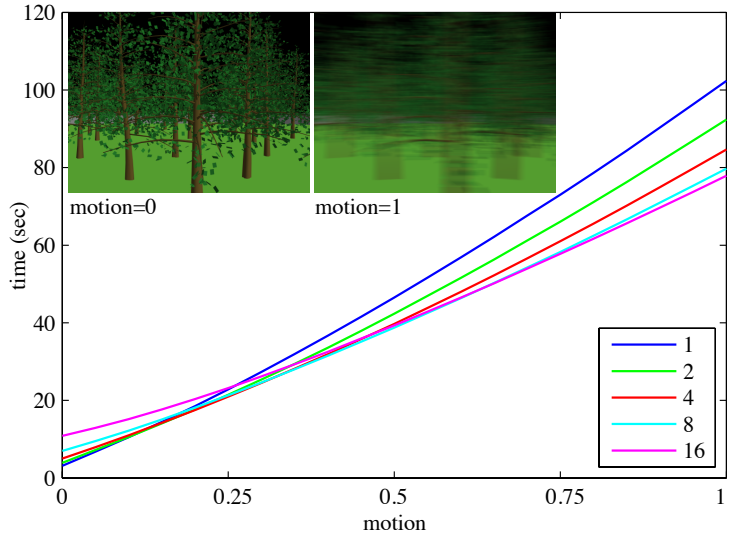


Figure 13: *Measurements from the trees scene, rendered with increasing panning motion. The different lines represent n = 1, 2, 4, 8 and 16.*

proximation generates high-quality images. This approximation is the reason why we choose to call our algorithm *semi-analytical*. Our algorithm can also be used for rendering of motion blurred ambient occlusion with high quality. In contrast to our work, the previous methods we are aware of for this are based on point sampling, and may require 512–1024 point samples for high quality for *static* scenes alone [26].

We believe our research opens up a wide range of interesting future work to be done. If point sampling is used in the spatial domain, we believe that our algorithm can be immediately used for computing semi-analytical (SA) depth of field, which is a two-dimensional problem. With our current algorithm, we can also approximate visibility over a lens with a set of line samples, which is similar to previous work [1], and also use line samples in the spatial domain. Extending with another dimension may be possible, and would open up for even more usage areas, such as SA depth of field with spatial line samples, or pixel area integration with SA motion blur, or SA depth of field with motion blur with spatial point sampling. While sampling patterns always are important in order to generate high-quality images, our focus has not been on developing high-quality line sampling patterns. Instead, our main focus has been on the creation of depth patches and the visibility engine. However, this is an interesting and important topic, and we will revisit this in future work. We would also like to integrate the occlusion culling more with the pixel integrator so that culling will work more efficiently for micropolygons.

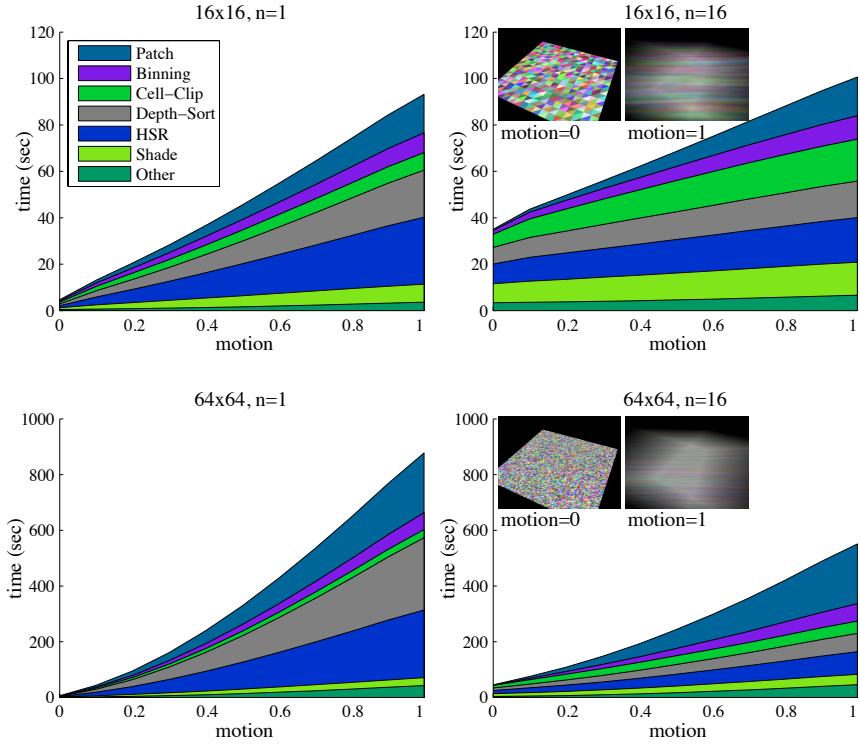


Figure 14: *Detailed measurements of single-core rendering of the plane scene with increasing panning motion. The different graphs represent various combinations of tessellation rate (*16×16 *and* 64×64*) and values for n (1 and 16). The "cell-clip" area corresponds to clipping of binned triangles to the cell boundaries, before depth sorting. The "other" area consists of vertex shading, BVH construction, BVH traversal, tile setup, line setup and tile to frame buffer transfer.*

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, the Swedish Research Council, and Tomas is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation. The Armadillo comes from the Stanford University Computer Graphics Laboratory.

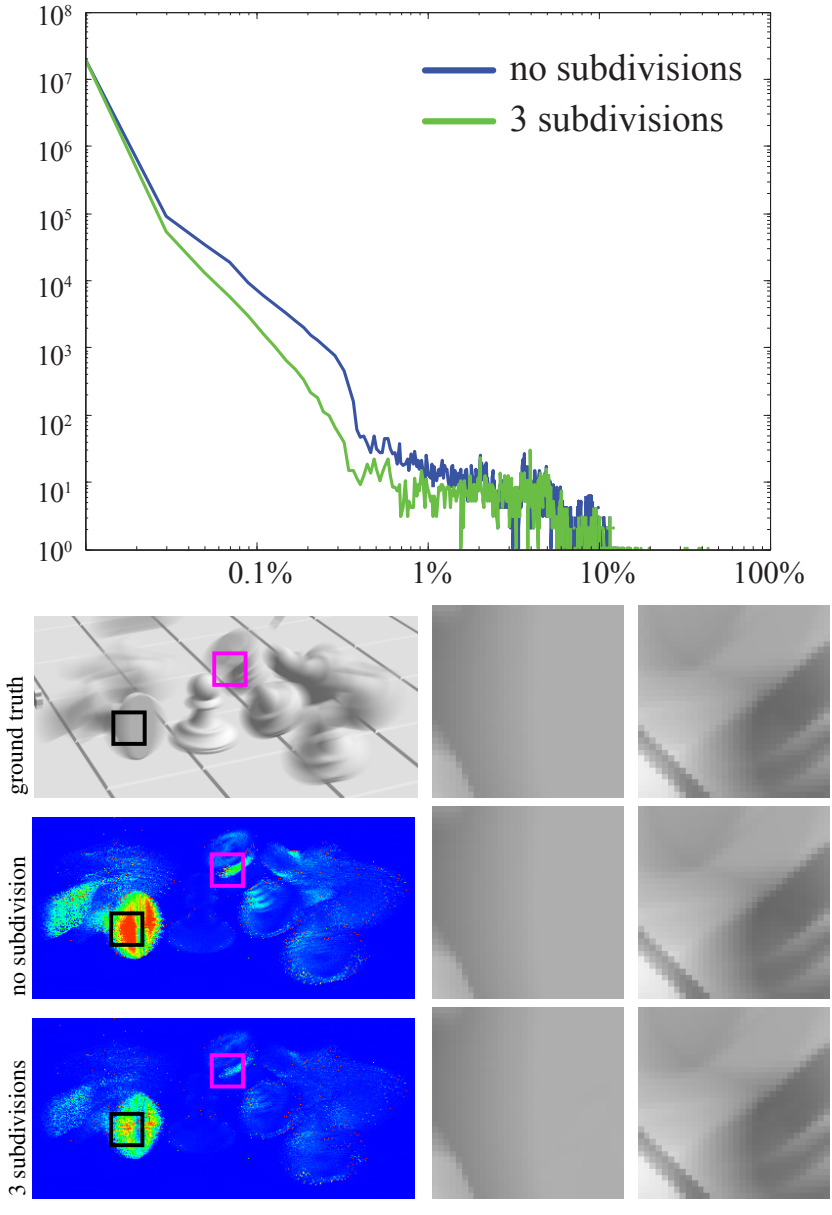


Figure 15: *Measurements and visualizations of introduced depth-approximation errors compared to a stochastic rendering. The loglog-histogram displays depth error as a fraction of the near- to far-plane distance. As is expected, the plots have identical integrals, even though this fact is somewhat obscured in the figure due to the logarithmic scale. Heat maps and zoom-ins of the scene then further detail distribution and magnitude (in log-scale) of the errors, as well as visual comparisons to the ground truth image.*

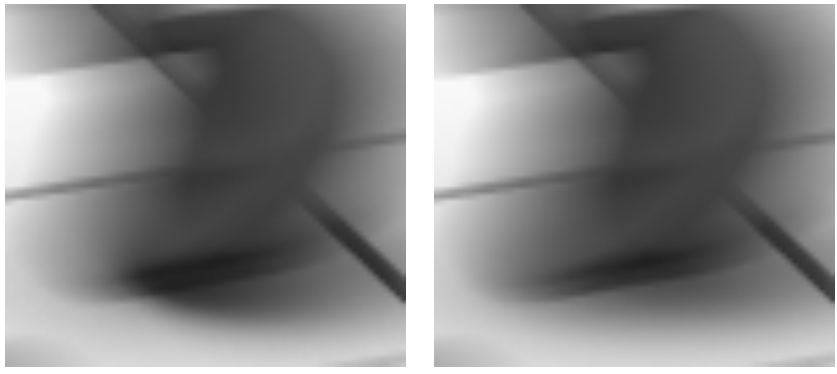


Figure 16: *A close-up of the topmost pawn in Figure 1, with static ambient occlusion (left) and motion blurred ambient occlusion (right). Both images were rendered with 16 line samples for ambient occlusion. Note that in cases like this, the difference is rather large.*

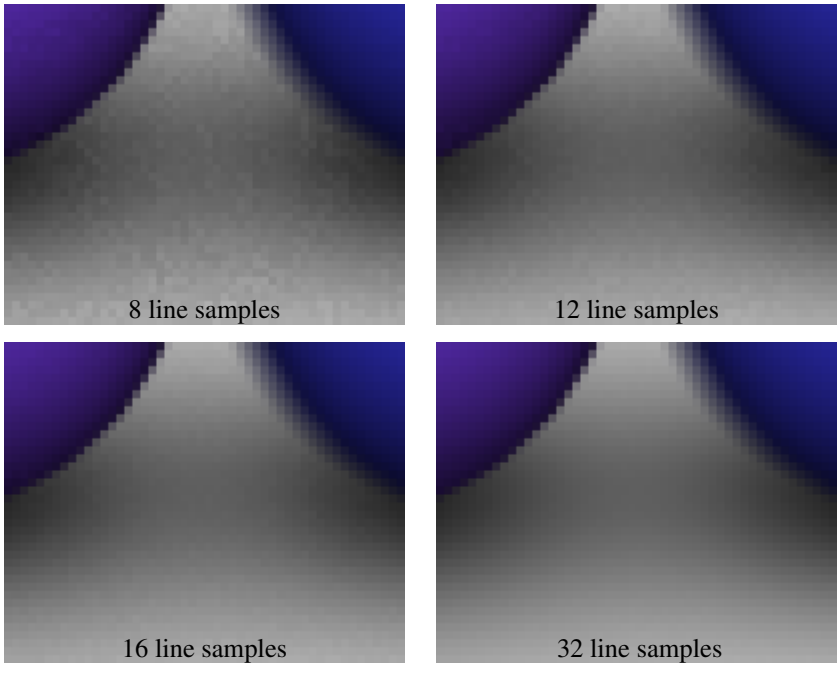


Figure 17: *In these images, we illustrate the effect of increasing the number on lines samples of the hemisphere for ambient occlusion.*

Bibliography

[1] AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware* (2007), pp. 7–16.

[2] BLINN, J. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications 12*, 4 (July 1992), 89–94.

[3] BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. A Lazy Object-Space Shading Architecture With Decoupled Sampling. In *High Performance Graphics* (2010), pp. 19–28.

[4] CATMULL, E. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[5] CATMULL, E. A Hidden-Surface Algorithm with Anti-Aliasing. In *Computer Graphics (Proceedings of SIGGRAPH 78)* (1978), pp. 6–11.

[6] CATMULL, E. An Analytic Visible Surface Algorithm for Independent Pixel Processing. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (1984), pp. 109–115.

[7] COOK, R. L., CARPENTER, L., AND CATMULL, E. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (1987), pp. 96–102.

[8] COOK, R. L., PORTER, T., AND CARPENTER, L. Distributed Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (1984), pp. 137–145.

[9] DALLY, W. Power Efficient Supercomputing. Accelerator-based Computing and Manycore Workshop (presentation), 2009.

[10] FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur. In *High Performance Graphics* (2009), pp. 59–68.

[11] FUCHS, H., KEDEM, Z., AND NAYLOR, B. On Visible Surface Generation by a Priori Tree Structures. In *Computer Graphics (Proceedings of SIGGRAPH 80)* (July 1980), vol. 14, pp. 124–133.

[12] GRABLI, S., DURAND, F., AND SILLION, F. Density Measure for Line-Drawing Simplification. In *Pacific Graphics* (2004), pp. 309–318.

[13] GRANT, C. W. Integrated Analytic Spatial and Temporal Anti-Aliasing for Polyhedra in 4-Space. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (1985), pp. 79–84.

[14] GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH* (August 1993), pp. 231–238.

[15] GRIBEL, C. J., DOGGETT, M., AND AKENINE-MÖLLER, T. Analytical Motion Blur Rasterization with Compression. In *High-Performance Graphics* (2010), pp. 163–172.

[16] HACHISUKA, T., JAROSZ, W., WEISTROFFER, R., K. DALE, G. H., ZWICKER, M., AND JENSEN, H. Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing. *ACM Transactions on Graphics 27*, 3 (2008), 33.1–33.10.

[17] HAEBERLI, P., AND AKELEY, K. The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (1990), pp. 309–318.

[18] HENNESSEY, J. L., AND PATTERSSON, D. A. *Computer Architecture: A Quantitative Approach*, 4th ed. MKP Inc., 2006.

[19] JONES, T. R., AND PERRY, R. N. Antialiasing with Line Samples. In *Eurographics Workshop on Rendering* (2000), pp. 197–205.

[20] KOREIN, J., AND BADLER, N. Temporal Anti-Aliasing in Computer Generated Animation. In *Computer Graphics (Proceedings of SIGGRAPH 83)* (1983), pp. 377–388.

[21] LAINE, S., AND KARRAS, T. Two Methods for Fast Ray-Cast Ambient Occlusion. *Computer Graphics Forum (Eurographics Symposium on Rendering) 29*, 4 (2010), 1325–1333.

[22] MANSON, J., AND SCHAEFER, S. Wavelet Rasterization. *Computer Graphics Forum 30*, 2 (2011), 395–404.

[23] MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. Hardware-Accelerated Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics* (2010), pp. 173–182.

[24] NAIMAN, A. C. Jagged Edges: when is Filtering Needed? *ACM Transactions on Graphics 17*, 4 (1998), 238–258.

[25] OWENS, J. D. Streaming Architectures and Technology Trends. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 457–470.

[26] PANTALEONI, J., FASCIONE, L., HALL, M., AND AILA, T. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. *ACM Transaction on Graphics 29*, 3 (2010), 37.1–37.10.

[27] RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics 30*, 3 (2011), 17:1–17:17.

[28] SUNG, K., PEARCE, A., AND WANG, C. Spatial-Temporal Antialiasing. *IEEE Transactions on Visualization and Computer Graphics 8*, 2 (2002), 144–153.

[29] SUTHERLAND, I. E., SPROULL, R. F., AND SCHUMACKER, R. A. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys 6*, 1 (1974), 1–55.

[30] WEILER, K., AND ATHERTON, P. Hidden Surface Removal using Polygon Area Sorting. In *Computer Graphics (Proceedings of SIGGRAPH 77)* (1977), pp. 214–222.

[31] WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. GPU-Accelerated High-Quality Hidden Surface Removal. In *Graphics Hardware* (2005), pp. 7–14.

[32] WHITTED, T. An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (1980), 343–349.

[33] ZHUKOV, S., IONES, A., AND KRONIN, G. An Ambient Light Illumination Model. In *Eurographics Workshop on Rendering* (1998), pp. 45–55.

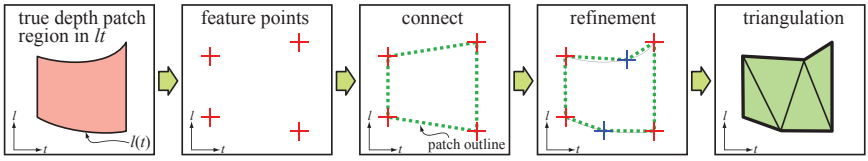


Figure 18: *Overview of depth patch triangulation.*

A Depth Patch Approximation

In this appendix, we will describe how our patch triangles, which approximate a depth patch, are generated. We start by deriving a formula for $l(t)$, which describes when a moving triangle edge intersects with the line sample plane. This function is needed to adaptively refine the approximation of the depth patch.

Recall that a vertex is described by $\mathbf{p}(t) = (1-t)\mathbf{q} + t\mathbf{r}$, $t \in [0,1]$. A moving edge, defined by $\mathbf{p}_0$ and $\mathbf{p}_1$, is then a bilinear patch: $\mathbf{b}(s,t) = (1-s)\mathbf{p}_0(t) + s\mathbf{p}_1(t)$. Without loss of generality, we assume that the line sample plane is $y = 0$, and that the l-parameter coincides with x. By setting the y-component of $\mathbf{b}(s,t)$ to 0.0, we can obtain an expression for s:

$$s = -\frac{p_{0y}}{p_{1y} - p_{0y}}, \tag{2}$$

where $p_{0y} = (1-t)q_{0x} + tr_{0x}$, etc. Replacing s by the equation above in the expression for the x-component of $\mathbf{b}(s,t)$, we obtain $l(t)$, after some simplification, as:

$$l(t) = \frac{p_{0x}p_{1y} - p_{0y}p_{1x}}{p_{1y} - p_{0y}}, \tag{3}$$

which clearly is a rational polynomial in t, where the numerator is of degree two, and the denominator is of degree one.

An overview of our depth patch triangulation algorithm is shown in Figure 18. Briefly, we first compute the *feature points* as described in Section 4. These are then connected as described below, and if needed, additional points are added adaptively in a refinement step. Finally, triangulation produces the final patch triangles.

Given a number of feature points in lt, each with a certain depth, d, the goal is now to form a triangulated approximation using these three-dimensional points. This is carried out by considering one moving edge at a time, and based on its interaction with the line sample plane as it moves over time, making connections between the feature points. The final set of connections outline one or many disjoint depth patches. In the final step, these patch outlines, shown in the middle in Figure 18, are triangulated.

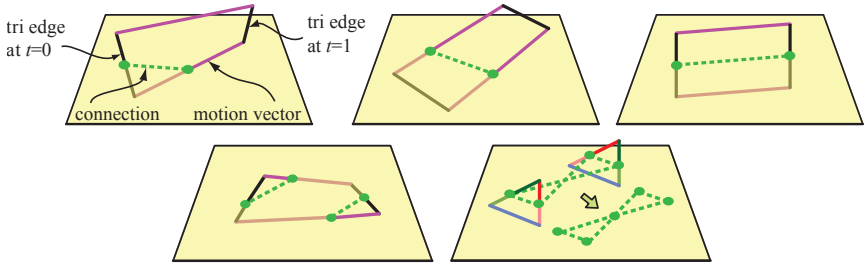


Figure 19: *Connections schemes for moving triangle edges. When there are two feature points (green circles), the feature points are simply connected. We show three possible situations that can occur (first three images). In the fourth image, the moving edge has generated four feature points. In this case, the two earliest (in t) feature points are connected, and then the remaining two. This assures that no connection will "bridge" the singularity point where the edge is parallel to the plane. The final image shows a situation where the facing of the triangle changes as it moves. Here, the connections will overlap in lt-space, and to resolve this, the patch is split by adding the intersection point.*

In the following, we describe how the feature points are *connected*. If the triangle at $t = 0$ or $t = 1$ intersects the line sample plane, one or two feature points are generated. In the case of two feature points, it is obvious that they should be connected and form an edge of the patch outline. However, we also observe that each moving edge may intersect the line sample plane, and trace out a curve in lt, and its corresponding feature points should also be connected to form part of the patch outline. If a moving edge generates two feature points, they must simply be connected. See Figure 19. When there are four feature points, the situation is a bit more complex, and the solution is illustrated to the right in the same figure.

Four feature points is an indication that the edge initially intersects the plane, turns parallel to it and then intersects the plane again in the opposite direction. At the point, t_s, of edge-plane parallelism, there will be no (edge is above or below plane) or infinitely many (edge is in the plane) intersection points, thus making $l(t)$ undefined for $t = t_s$. Two feature points will reside on each side of t_s, that is, at $\leq t_s$ (initial intersection) and $\geq t_s$ (second intersection), respectively. By connecting the feature points on each side of t_s, we avoid letting any connection span undefined values of t. In the extreme, when the edge is precisely in the plane at t_s, two feature points will occur at t_s, and will thus constitute end point for one of the connections and start point for the other. Hence, each connection will span an interval (t_a, t_b) (not including the end points) that is defined in $l(t)$.

After all connections have been made, we have a set of lines, and we simply create the patch outline (middle in Figure 18) by connecting lines that share feature point vertices. In a final step, the patch triangles are created by triangulating the resulting polygon(s). Our approach to generate the patch triangles works really well for

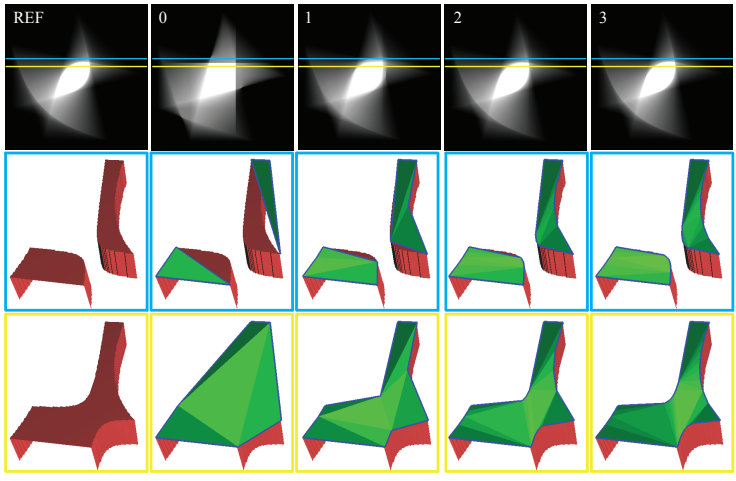


Figure 20: *At the top left, we show a rotating white triangle rendered with 625 stochastic samples, i.e., a reference image. Below that image, we also show the corresponding depth patches (generated using dense point sampling) for the yellow and blue lines in the image. For the top line sample, note how the line will reside completely outside the triangle, during a short interval, which generates two disjoint depth patches (shown in the middle row).*

triangles that undergo moderate rotation, however, the approximation may be too crude for extreme motion. To that end, we present an adaptive refinement procedure.

A.1 Adaptive Refinement of the Approximation

Better approximation accuracy can be achieved by adding more points to the patch outline before triangulating. This is done by exploiting that $l(t)$ (Equation 3) is defined, and monotonic in $\frac{\partial l}{\partial t}(t)$, within each connection range (t_a, t_b). Hence, additional points can be created by evaluating $l(t)$ at arbitrary values of $t \in (t_a, t_b)$. A good choice of t is one that captures as much of the curvature of $l(t)$ within the range as possible. In our approach, we interpolate the angle of the tangent slopes in the end-points of the range, then solve $\frac{\partial l}{\partial t}(t)$ for this value to retrieve t, which we use to generate a new point on the patch outline. If just one extra point is to be obtained, the average of the two angles is used, and the theory goes as follows. Given an interval (t_a, t_b) for a connection, the angles, α_a & α_b, of the slopes at t_a and t_b are:

$$\alpha_i = \arctan\left(\frac{\partial l}{\partial t}(t_i)\right), \text{ for } i = \{a, b\}. \tag{4}$$

Using the arithmetic mean of the angles, we then need to solve the following equation for t:

$$\frac{\partial l}{\partial t}(t) = \tan\left(\frac{\alpha_a + \alpha_b}{2}\right) \tag{5}$$

Since $\frac{\partial l}{\partial t}(t)$ is quadratic in t, this will result in up to two solutions, but only one can be part of (t_a, t_b). By evaluating $l(t)$ for this new t, a new approximation point is acquired. For additional points, further subdivision can be made adaptively. Examples of termination criterion are maximum number of new points, depth error tolerance etc. As the number of subdivisions grows toward infinity, the approximation outline approaches the analytical solution in lt-space. In Figure 20, we show a triangulated depth patch for different levels of subdivision. Note, however, that the depths will still be linearised over the interior of this outline.

B Shading Cache

We use an object-space shading cache [3] to avoid excessive shading, and we extend their cache with an image pyramid for efficient memory usage. Note that we use the same cache for all our rendering threads, and hence need to allocate 2.0 GB for the entire cache for our renderings. Each object's shading values are stored in a map, which is represented by 16×16 independent submaps, where each submap is responsible for a region of the uv-space. Each submap consists of a pyramid of cached shading values, which is allocated lazily. At the top of the pyramid, a single shading sample represents the entire uv span of the submap. This is the lowest shading resolution available. Each lower level in the pyramid has double the resolution of the previous level.

We use the derivatives of the barycentric coordinates, u and v, with respect to screen space, x and y, when selecting appropriate pyramid levels. This is analogous to standard mipmapping. When a shading request is processed, the correct pyramid level is selected based on the these derivatives, and then the uv-coordinates of the shading request is sampled using nearest neighbor. If the cache contains a shaded value, it is immediately returned and used. If the cache location is empty, the pixel shader is invoked and the result put into the lowest level of the pyramid and propagated upwards. During upward propagation, the higher levels of the pyramid are sampled at the same uv-coordinates. If the higher level sample is empty, the same cached value is put into that pyramid level, and propagation continues. If the higher level sample is set, the propagation is aborted. When multiple threads operate on the same shading cache, sampling and propagation are subject to race conditions. We use atomic writes to the shading cache which makes it consistent, even during propagation. Without a shading cache, our system will perform far less shading requests than our stochastic rasterizer. The shading cache was added both for improving performance and to make the comparison with stochastic rasterization more fair.

Paper II

High-Quality Curve Rendering using Line Sampled Visibility

Rasmus Barringer Carl Johan Gribel Tomas Akenine-Möller

Lund University

ABSTRACT

Computing accurate visibility for thin primitives, such as hair strands, fur, grass, at all scales remains difficult or expensive. To that end, we present an efficient visibility algorithm based on spatial line sampling, and a novel intersection algorithm between line sample planes and Bézier splines with varying thickness. Our algorithm produces accurate visibility both when the projected width of the curve is a tiny fraction of a pixel, and when the projected width is tens of pixels. In addition, we present a rapid resolve procedure that computes final visibility. Using an optimized implementation running on graphics processors, we can render tens of thousands long hair strands with noise-free visibility at near-interactive rates.

ACM Transactions on Graphics, 31(6):162:1–162:10, 2012.

Figure 1: *Our novel thin curve rendering algorithm used on a test production model to compute accurate visibility. The model has 32,000 unique hair strands, which consists of over one million Bézier curves with varying thickness. As can be seen, our algorithm works at all different scales, from cases where there are hundreds of hair strands per pixel to zooming in on the hair strands. All images were rendered at* 1024×1024 *pixels with our GPU implementation. The leftmost image took 109 ms to render, while the close-up on the face took 468 ms. The rightmost image showcases our ability to handle thick curves. Hair model courtesy of Weta Digital.*

1 Introduction

High quality rendering of thin, curved primitives, e.g., hair, fibers, fur, and grass, is an important ingredient in today's computer generated imagery. This is particularly true for offline rendering for feature films, but also increasingly so for real-time rendering in games. One approach is to model such thin curves as ribbons with varying width, e.g., using RenderMan's `riCurves` primitive, and then sample visibility using point sampling. A similar modeling and rendering technique was used by Marschner et al. [21] when developing an accurate appearance model for hair. Another common approach is to rasterize lines with alpha blending to simulate line widths smaller than one pixel [19, 27]. A third approach is to model and render hair with volumetric textures using ray marching [16].

While shading for some types of thin primitives, in particular hair [21, 24, 33, 31, 14], is well understood, computing accurate visibility rapidly for a large number of curves remains a challenge. A major problem is that when point sampling is used, noise is inevitable unless a very large number of samples per pixel is used. This is especially true at a macro-scale, when the viewer is relatively far away from the curves, and the projected width of the curve is only, say, 10% or less, of the pixel width. In such cases, hundreds of samples per pixel may be needed for accurate visibility. For comparison, the diameter of a hair strand is about 0.1 mm [12]. Another problem is that the ribbon model breaks down at the microscale, i.e., when a curve's width project to relatively large number of pixels. In those cases, a hair strand, for example, does not appear as a cylinder as expected.

As a solution to this challenge, we present a visibility engine based on line sam-

pling [15] in the spatial domain. Our curves are modeled as Bézier splines with varying thickness. We develop a novel intersection algorithm between such curves and line samples and present a new interval resolve procedure. As can be seen in Figure 1, our approach renders practically noise-free images at large spectrum of scales. For rapid rendering, we have also implemented our visibility engine on a graphics processor running in parallel.

2 Previous Work

There is a wealth of literature on the topic of simulation, animation, and rendering of thin curves, e.g., hair strands, fur, and grass. Here, we will review the most important references to our work, and refer to the SIGGRAPH course [12] on hair and strands if more information is needed.

Appearance Models Kajiya and Kay [16] presented an illumination model for single-scattering in fur and hair. The fine geometric detail of fur and hair is represented by volumetric three-dimensional textures. Illumination is calculated by integrating diffuse and specular contributions, using a modified Phong model, by ray marching in the texture. Marschner et al. [21] present a rich shader model for human hair that captures more distinguished features such as inner reflection, eccentricity and surface scales. Fibers are represented geometrically as procedurally generated flat ribbons and as transparent elliptic cylinders within the shader model. This work was extended to include multiple scattering of light using photon mapping [23]. Rendering was made using two passes: first particles were traced through the hair to create a photon map, and then the hair was ray-traced to compute direct illumination and indirect radiance gathered from the photon map. Moon et al. [24] voxelize the individiual fibers into a rectilinear grid, and populate it with aggregate data of nearby fibers using sampling of density, median direction, and variance. By taking advantage of the smoothness of the distributed scattering function, this approach proved faster and less memory-consuming than approaches based on photon mapping. Zinke and Weber [32] present a shading framework for fibers that have both near and far field solutions. Recently, Hery and Ramamoorthi have presented an importance sampling algorithm for hair fibers [14]. Fast, but accurate models for shading hair using a dual scattering approximation, targeting real-time rendering, have also been presented [33, 31].

Visibility and Shadows The deep shadow mapping algorithm [20] generates a monotonic function per shadow map texel, which represents the fractional visibility for each depth from the light source. This works particularly well for hair, fur, and smoke. Jones and Perry [15] presented line sampling in the spatial domain for efficient anti-aliasing. Gribel et al. [11] used a similar approach applied to analytical motion blur with spatial point samples, instead of using it for spatial line sampling with no motion blur. Recently, that method has been combined with spatial line samples, for high-quality spatio-temporal anti-aliasing [10]. Tzeng et al. [29] used line samples over the lens domain to create depth-of-field. A common

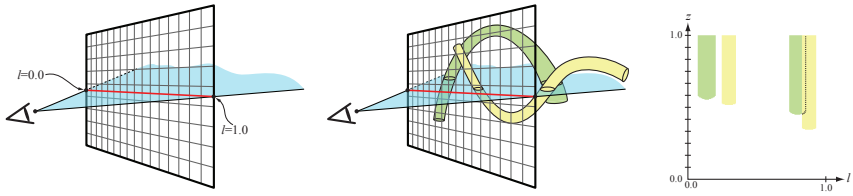


Figure 2: *Left: a line sample is defined by the camera location and a line (red) in screen space, and parameterized by l. Middle: two curves with thickness intersect the line sample. Right: visibility is resolved in the lz-plane by finding the closest surfaces in depth, z, along l. As can be seen, the yellow thin curve occludes part of the green thin curve over this line sample to the right, but not to the left.*

rendering technique using graphics processors, is to render the hair strands as lines with alpha blending [19], and composite the fragments in back-to-front order [27]. An alternative is to use stochastic transparency, where multi-sampled anti-aliasing is used to represent transparency in a pixel [8].

Offsets and Subdivision Our intersection algorithm is based on offsets for Bézier curves and subdivision. The offset to a polynomial curve is *not* polynomial in general [7]. A standard approach is subdivide the curve until the offset of each subsegment can be represented by a simpler primitive such as a line or curve. This can be done recursively [3] (recursive subdivision) or more sophisticated rules can be used in order to reduce the number of subdivisions [13]. A comparison of different curve offset methods is presented by Elber et al. [7]. Tiller and Hansen use quadratic Bézier curves to construct the offset curve, where the edges of the control polygon are offset [28]. A method suited for non-constant curve radius, is to offset each control point in their respective normal direction [4]. Recently, Ruf [26] presented a method for fast creation of quadratic bounding-Bézier offsets of quadratic Bézier curves.

3 Algorithm Overview

In this section, we present a high-level overview and motivation of some design choices of our visibility algorithm for high-quality rendering of thin primitives, and we also introduce key terminology used throughout this paper.

To sample visibility, we rely on *line sampling* [15] rather than the commonly used point sampling. There are several appealing aspects of the utilization of line samples when rendering a scene. As noted by Gribel et al. [10], compute power rather than memory bandwidth is used to a greater extent, which meshes well with recent trends in hardware development. Furthermore, sampling using lines effectively means that scene geometry is considered through an entire dimension instead of at discrete points, which means that there is a smaller risk of missing thin primi-

tives with line samples. As illustrated to the left in Figure 2, a line sample can be thought of as the triangle defined by the location of the camera and by a line in screen space, where the triangle extends towards infinity beyond the image plane. We will refer to these as *line sample planes* or simply as *line samples*, where each line sample is parametrized by $l \in [0,1]$. Usually, a small number of line samples per pixel is required to faithfully sample visibility.

In general, a thin primitive is defined by a three-dimensional Bézier spline, which represent the "core" of, say, a hair strand. In addition, a two-dimensional profile is swept along the curve, and this traces out the geometry that we need to compute visibility for. We call the generated geometry a *thin curve*, for the lack of a better term. A circular profile generates a shape similar to sweeping a sphere along a curve [30], and it is also related to computing offset curves. See, for example, the recent work by Ruf [26]. In all our examples, we will sweep a circular profile whose radius varies along the curve, but this can be extended to other profiles.

In broad terms, our algorithm works as follows. For parallel execution of our visibility algorithm, we employ a *sort-middle* approach [22], where the geometry is binned to rectangular pixel tiles. The geometrical content within each tile is binned once more to line samples in the tile. Next, the actual intersections between the curves and their associated line samples are computed. This is illustrated in the middle in Figure 2. The intersection is approximated by a set of *intervals* defined by a start point and an end point in lz-space, where z is depth. When all intervals for a line sample in a tile have been identified, visibility has to be resolved by finding the nearest (in depth) interval segments along the entire line sample. To that end, we present a simple, yet very efficient resolve procedure, which is substantially faster than the resolve presented by Gribel et al. [11]. An example is shown to the right in Figure 2. We store intervals in a binary heap ordered according to starting points, and resolve visibility in one pass without extra sorting and no extra storage. Shading is computed by taking point samples over the intervals. In the end, the colors from all line samples of a pixel are weighted to get the final color of the pixel.

Next, we present our visibility engine, which includes a description of our geometry representation, intersection computations, and our new resolve procedure.

4 Visibility Engine

In this section, we describe all the involved components and algorithms in our visibility engine for high-quality thin curve rendering. First, our representation of thin curves is presented, and then follows our intersection algorithm between a line sample and a thin curve in Section 4.2, which is extended in Section 4.3 to handle situations where the projected width of a curve covers many pixels. Finally, our novel resolve procedure is presented in Section 4.4.

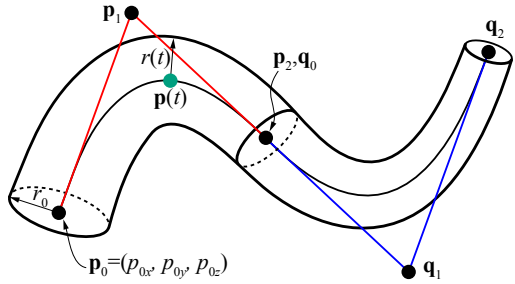


Figure 3: *A thin curve composed of two quadratic Bézier curves. A circular profile is used in this example, and each control point consists of a three-dimensional position, and a radius of the circular profile. The spatial Bézier curve to the left is defined by* $\{\mathbf{p}_i\}$*, while the one to the right is defined by* $\{\mathbf{q}_i\}$*,* $i \in \{0,1,2\}$*. Note that to ensure* G^1 *continuity, we set* $\mathbf{p}_2 = \mathbf{q}_0$*, and also make sure that the tangent directions at that shared position is the same on both sides.* C^1*-continuity is obtained if the tangents also have the same length.*

4.1 Thin Curve Representation

We represent a thin curve by a quadratic Bézier spline, which is a series of connected quadratic Bézier curves. In addition, the "thickness", or radius, varies along our thin curves. A quadratic Bézier curve segment is defined using three control points [9]. We use both spatial control points, $\mathbf{p}_i$, as well as radii, r_i, $i \in \{0,1,2\}$, to define the curve's position, $\mathbf{p}(t)$, $t \in [0,1]$, and interpolated radius, $r(t)$:

$$\mathbf{p}(t) = \sum_{i=0}^{2} B_{i,2}(t)\mathbf{p}_i, \quad r(t) = \sum_{i=0}^{2} B_{i,2}(t) r_i, \tag{1}$$

where $B_{i,2}(t)$ are Bernstein basis polynomials. The varying radius makes it possible for, e.g., a piece of fur to be thicker at the animal body and thinner towards the other end. Note that the degree two of the Bézier curves was chosen to make the intersection algorithm simpler, but in principle, a higher degree can be used at the cost of a more involved intersection algorithm.

In general, the radius controls the size of a two-dimensional profile that is used to define the character of the thin curve. In all our examples, we use circular profiles. The geometry of the thin curve is defined by sweeping the two-dimensional profile along the Bézier curve while varying the size of the profile according to the interpolated radius, $r(t)$. This is illustrated in Figure 3. Instead of a circular profile, one can use two connected straight lines, e.g., a V, which could be used to model grass, for example.[1] Exploring different profiles and developing their corresponding intersection tests is left for future work. It is important to ensure (at least) G^1 continuity when connecting neighboring Bézier curves. Any global optimization

[1] This would require extending each control point with a binormal as well in order to establish a coordinate system for each t. This is not needed for circular profiles since they are fully symmetric.

methods [9] can be used for this, or simpler heuristics as desired by the user.

In Appendix A, a memory-efficient representation is presented. Next, we present our intersection method between a line sample and a thin curve.

4.2 Thin Curve/Line Sample Intersection

Our visibility engine requires us to compute the intersection between thin curves and line sample planes. Our approach to solving this efficiently is based on using *offsets* [7, 28, 4, 26]. We will use *approximate* offsets, and intersect these with the line sample planes using a subdivision technique to reach a certain error tolerance. The offset intersections are then projected to screen space (along the sample line), connected into intervals, and fed to the visibility engine.

Offsets

The offset, $\mathbf{o}(t)$, of a curve can be said to represent all points being located at a distance, $r(t)$, in the normal direction of $\mathbf{p}(t)$. The normal is defined as $\mathbf{n}(t) = \frac{\mathbf{p}'(t)\times\mathbf{v}(t)}{|\mathbf{p}'(t)\times\mathbf{v}(t)|}$, where $\mathbf{p}'(t)$ and $\mathbf{v}(t)$ are the curve tangent and view-vector, respectively. In such a setting, the two offset curves of $\mathbf{p}(t)$ can be described as:

$$\mathbf{o}^{\pm}(t) = \mathbf{p}(t) \pm r(t)\mathbf{n}(t). \tag{2}$$

A consequence of the presence of the square root in the expression for $\mathbf{n}(t)$ is that, even though $\mathbf{p}(t)$ is in polynomial form, its offsets $\mathbf{o}^{\pm}(t)$ are, in general, not [7]. Hence, to find the intersections efficiently, we use approximations, $\mathbf{o}_a^{\pm}(t)$, of the real offsets $\mathbf{o}^{\pm}(t)$. The offset approximation by Cobb [4] is defined by the control points of the original curve, offset by a constant amount, r, in their respective normal direction, $\mathbf{n}_i$. Since we have varying radius along the curve, we use the following, slightly modified version of that offset approximation:

$$\mathbf{o}_a^{\pm}(t) = \sum_{i=0}^{2} B_{i,2}(t)(\mathbf{p}_i \pm r_i\mathbf{n}_i), \tag{3}$$

where $\mathbf{n}_0 = \mathbf{n}(0)$, $\mathbf{n}_1 = \mathbf{n}(0.5)$, $\mathbf{n}_2 = \mathbf{n}(1)$, $r_0 = r(0)$, $r_1 = r(0.5)$, and $r_2 = r(1)$. The two offset curves, $\mathbf{o}_a^{+}(t)$ and $\mathbf{o}_a^{-}(t)$, are illustrated in Figure 4 for several examples. Intersecting Equation 3 with a plane yields a second degree polynomial, and it is therefore fast and simple to compute these intersections. However, similar to Cobb's method, it will produce approximate offsets that, in general, underestimates the real offset. Given a relative error tolerance, ε, the offset approximation, $\mathbf{o}_a(t)$, will meet this tolerance at a certain t, if the following expression holds:

$$r(t)^2(1-\varepsilon)^2 \leq (\mathbf{o}_a(t) - \mathbf{p}(t))^2 \leq r(t)^2(1+\varepsilon)^2, \tag{4}$$

which simply is a more efficient calculation of the relative error test: $\|1 - \|\mathbf{o}_a(t) - \mathbf{p}(t)\|/r(t)\| \leq \varepsilon$. To obtain arbitrary precision, we use a subdivision process, which is described next.

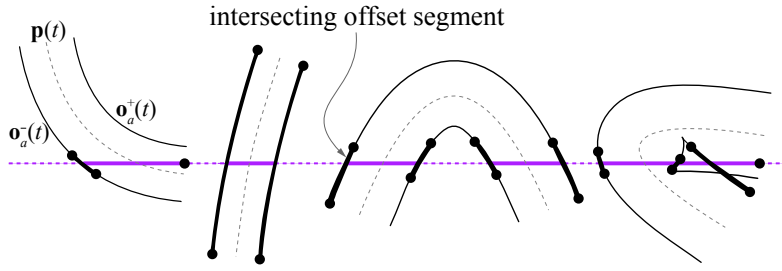


Figure 4: *A horizontal line sample (purple) is shown together with a number of quadratic Bézier curves, $\mathbf{p}(t)$, with thickness. The two offset curves, $\mathbf{o}_a^+(t)$ and $\mathbf{o}_a^-(t)$, of $\mathbf{p}(t)$ are also shown. Since the offset curves are approximate, we use a subdivision approach to refine the offsets until the error is sufficiently small. First, an offset is created from $\mathbf{p}(t)$, and then, depending on curvature, it is subdivided in order to reach the desired error tolerance. The second curve from the left has modest curvature and requires no subdivision at all, while the rightmost, which is strongly curved and even contains a self-intersecting loop, requires several subdivisions. Such cases are very rare in practice. Once one or multiple satisfactory offset segments (fat black curves) are found, they are intersected with the line sample to produce roots outlining the silhouette of the intersection (solid horizontal lines).*

Offset Creation by Subdivision

The approximate offsets, $\mathbf{o}_a^{\pm}(t)$, are created by splitting $\mathbf{p}(t)$ through subdivision into shorter segments. As the segments get shorter, the offset approximations become more accurate. Hence, accuracy is traded for increased cost of subdivision and offset generation. The nature of the approximation in Equation 3 implies that the end-points of the offset approximation will correctly match the real offset, and subdivision will therefore ensure convergence towards the real offset due to the end-point interpolation property of Bézier curves [9].

To avoid unnecessary work, we cull away and abort subdivision for offset segments that will not end up intersecting the line sample. Since the control point triangles of $\mathbf{o}_a^{\pm}(t)$ do not necessarily bound the real offset, these control point triangles cannot be used for this purpose. Instead, we cull against the *offset bound* of $\mathbf{p}(t)$, which we define as the convex hull of circles of radius, r_i, located at the control points, $\mathbf{p}_i$. Once an offset bound is above or below a line sample, the corresponding offset curve can be culled safely.

To be as memory efficient as possible, the algorithm subdivides successively by maintaining a stack of t-intervals that are eligible for intersection. The starting element of the t-stack is set to $\Delta t = [0,1]$, meaning that the entire offset is eligible initially. We will use subdivision [9] of the Bézier curve in our algorithm, and will

use the notation $\mathbf{p}_i^{\Delta t}$ for the control points of a subdivided Bézier curve, derived from $\mathbf{p}_i$, which is valid over Δt. Given a curve $\mathbf{p}(t)$, defined by $\mathbf{p}_i$, and a line sample, L, the algorithm has the following steps, applied to one side (either $\mathbf{o}_a^+(t)$ or $\mathbf{o}_a^-(t)$) of the offset at a time:

1. Pop the top element, Δt, from the t-stack, and subdivide the curve using De Casteljau's algorithm to obtain $\mathbf{p}_i^{\Delta t}$. If the t-stack is empty, then exit algorithm.
2. If L does not intersect the offset bound of $\mathbf{p}_i^{\Delta t}$ then goto 1.
3. Create approximate offset to $\mathbf{p}_i^{\Delta t}$ according to Equation 3.
4. If the error test, according to Equation 4, of the approximation is fulfilled: compute intersections between the approximate offset and L and project them to screen space. Else, split Δt into two halves and push to the t-stack. Goto step 1.

In a final step, intersections between the end-cap lines of the thin curve and L are computed. The output from the intersection algorithm for a complete curve, including both offset curves and the end-caps, is up to four intersection points. See Figure 4.

The termination criteria in step 4 with the user-defined error tolerance, ε, is preferably complemented with a maximum subdivision depth, N_{max}. The t-stack will then contain at most $2N_{max}$ elements, and each element Δt can be stored using $2 \cdot 16$ bits of memory. In addition to the storage for the stack, storage is needed for the current subdivided control points, $\mathbf{p}_i^{\Delta t}$, its offset approximation, and the roots in t, generated from solving the second degree polynomial. The memory requirements are thus bounded and relatively small. For practical reasons, we evaluate the error function, $e(t)$, in Equation 4 at two discrete positions spread uniformly over the interval Δt. The end points of Δt are not included since the error is zero there. As a heuristic, we require that the error tolerance is to be met at these two locations before terminating the subdivision. For thin curves, we have observed that tolerances of $\varepsilon = 0.5\% - 1\%$ are typically sufficient. The left part of Figure 5 illustrates convergence of the offset by subdivision for a heavily bent curve.

Next, the intersection points are connected into intervals using a few simple heuristics. If only two intersections are present, they are obviously connected and we are done. For four intersections, there will be two intervals that are either disjoint or overlapping, depending on the curve parameter, $t \in [0,1]$. See Figure 4. The intervals are formed in a disjoint manner if the first two and last two intersections along L are disjoint with respect to t (third example in Figure 4). However, if the t-spans of the first pair and last pair overlap, the intervals are consequentially formed to overlap, simply by connecting the first intersection point with the third, and the second with the fourth (rightmost example in Figure 4). Overlapping intervals is a result of curves that undergo self-intersection. Self-intersecting curves, such as the right curve in Figure 5, may in fact give rise to as many as six intersections

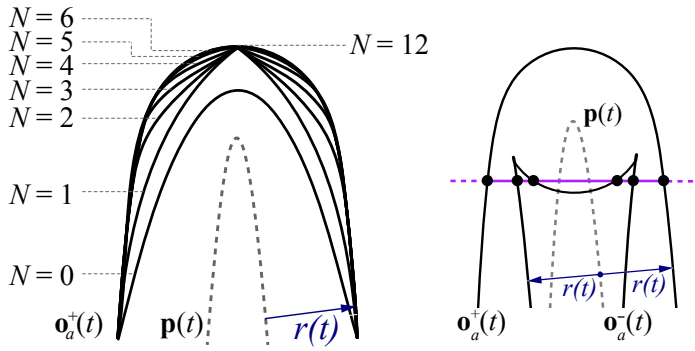


Figure 5: *Left: evolution of curve offset through subdivision. An offset curve, $\mathbf{o}_a^+(t)$ is created by displacing the control points of the original curve, $\mathbf{p}(t)$, in the normal direction. With no subdivision, $N = 0$, the offset always underestimates the true offset. As N increases, the core curve is split into smaller and smaller segments, and the accuracy of the offset increases. Right: Up to six intersections may arise along a line sample in cases with self-intersection. Here, only four are kept: two from the outer offset, and the outermost two from the inner offset.*

along the sample line. For simplicity, our algorithm keeps only the outermost two, limiting the number of intersections to four for the entire curve.

Though self-intersection may seem somewhat abstract and unintuitive since real-world objects usually do not self-penetrate [28], it is nevertheless commonplace in our setting as curves are projected from 3D to 2D. In other words: a curve with no self-intersection in 3D-space may self-intersect when projected to 2D from a certain point of view. It is for this reason necessary for our algorithm to robustly handle all kinds of curve behavior.

4.3 Visibility for Large Projected Curve Widths

When the projected width of a curve covers many pixels, it should become possible to see the geometrical shape of the curve. For example, a hair strand should look like a curved cylinder. The algorithm in Section 4.2 only gives the endpoints of an intersection between a thin curve and a line sample. When the projection is rather large, this is not sufficient. Here, we present a simple extension which solves this case.

Recall that the curve is described by a Bézier curve, $\mathbf{p}(t)$, and a radius, $r(t)$. For circular profiles, this can be interpreted as a circle with varying radius that moves along the curve, $\mathbf{p}(t)$, as a function of t, and we need the intersection between the thin curve and the line sample plane. This intersection consists of one or more closed curves.

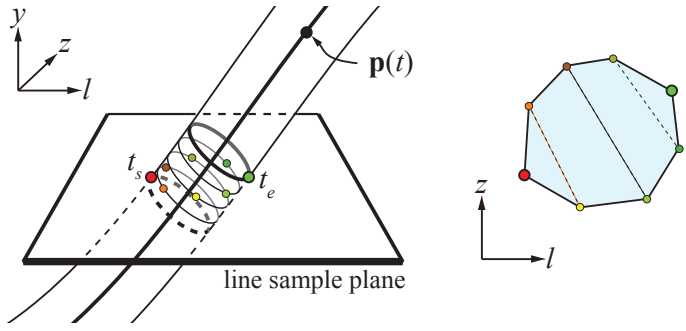


Figure 6: *To compute the intersection between a curve whose width projects to many pixels, and a line sample plane, the time t_s and t_e are first computed. These are the times when the moving circle (as a function of t) first intersects the line sample plane, and when it exits. In this case, we generate three extra circles between t_s and t_e, and all circles are intersected against the line sample plane. Together these form a tessellated approximation (right) to the true intersection.*

To this end, the normal used for thin curves in the previous section is replaced by an expression that generate offset curves above and below the thin curve along the y-axis:

$$\mathbf{n}(t) = \frac{\mathbf{p}'(t) \times \big(\mathbf{p}'(t) \times (0,1,0)\big)}{|\mathbf{p}'(t) \times \big(\mathbf{p}'(t) \times (0,1,0)\big)|}. \tag{5}$$

Next, we use the method from the previous section to compute the first time, t_s, where the moving circle touches the line sample plane, and the time, t_e, when the moving circle exits the line sample plane. This is illustrated in Figure 6. Given the time interval, we essentially use a tessellation procedure to compute an approximation of the true intersection curve, which, in general, is a complex high-order curve. First, n uniform t-values between t_s and t_e, are computed, i.e., $t_i = (1-\alpha_i)t_s + \alpha_i t_e$, $\alpha_i = i/(n-1)$, where $i \in \{0,1,\ldots,n-1\}$. In our implementation, we use $n = 32$. However, any number can be used, and it may be beneficial to calculate a suitable number based on the thickness of the curve. Next, n circles are generated centered at $\mathbf{p}(t_i)$, with radius $r(t_i)$, and the normal of the plane equation in which the circle lies is $\mathbf{p}'(t)$, i.e., the tangent of the curve. These circles are intersected with the line sample plane, and the points connected to form a closed tessellated curve. This is shown to the right in Figure 6. The lines of this tessellation are inserted (as usual) as intervals into our visibility engine. An example of possible results using this technique is shown in Figure 7.

As discussed previously for thin curves, a single curve can create multiple disjoint intersections with a line sample. One such example is the third curve in Figure 4. Self-intersecting curves, such as the right curve in Figure 5, exhibits a similar behavior except that the two intervals overlap. For thick hairs, each such interval is tessellated independently of others.

Figure 7: *An extreme example of thin curves whose widths project to a large number of pixels (about* 20 *at the base). Note that each hair strand's width transitions to subpixel size towards their outer end points.*

4.4 Interval Resolve Procedure

As described in the previous subsections, the intersection computations between thin curves and a line sample generate a list of intervals. In order to determine the visibility along a line sample, and ultimately determine the color of the pixels overlapping the line sample, we need to find the (clipped) intervals closest in depth to the camera. By replacing t (time) for l (the parameter along the line sample), it would be possible to use the resolve procedure by Gribel et al. [11]. However, we have devised a novel resolve algorithm, specialized for opaque geometry, which is simpler, uses less memory, and is therefore faster. Our algorithm is described below.

All intervals are stored in a binary heap [6], ordered by their starting interval point. Note that the intervals are not sorted and we simply perform a build heap operation which takes $O(n)$ time. This enables us to do efficient insertions, and removals, while avoiding the dependent memory accesses inherent in any kind of self-balancing search tree. Our algorithm performs a single sweep over the line sample and process two intervals at a time. For each pair, the space that separates them can be resolved immediately and accumulated to the exposed pixels. The sweep continues with the interval closest to the camera. If the occluded interval spans past the point of occlusion, it is reinserted into the heap at a point where it may be visible again. The details are shown in Algorithm 1. An illustration of how our resolve works in a simple situation is shown in Figure 8.

The most expensive parts in the resolve procedure are the removal from and the

insertion to the heap. As such, it is very important to realize that these operations can be performed simultaneously. We can read the minimum element from the heap by inspecting the first value in its array. Removal moves the last element in the array to the first position and then restores heap order. Insertion puts the new element last and restores heap order. By delaying the removal of the minimum element until the very end of the loop, we can choose whether to replace the minimum element by the interval that is to be reinserted, or by the last element of the heap. We can therefore get away with a single restore of heap order. This also ensures that instruction divergence is kept to a minimum since we only need a small branch deciding what element to put first in the heap. This property is very important for GPUs.

Note that each interval is stored only once in the heap which means that given n intervals, we need to store only n items. The memory requirements are thus predictable as they do not depend on the geometric relationship between intervals. Given n intervals, the previous resolve [11] needs to store at least $2n$ items, and in the worst case, $\frac{1}{2}(3n+n^2)$ items depending on the number of intersections. Another benefit of our algorithm is that we only intersect intervals against the interval closest to the camera, resulting in far fewer intersections than when finding all intersections between all intervals. The number of intersections are kept to a minimum since our algorithm includes a form of occlusion culling. When an interval is (partially) occluded, it will be discarded from all calculations until a point where it may be visible again (if such a point exists). When the interval is no longer occluded by the previous occluder, it is checked again against the interval closest to the camera. If it is still occluded, it will be discarded again. An interval can thus be efficiently occlusion culled even if it is only occluded collectively by multiple intervals.

The algorithm shown here includes intersections between intervals for completeness only. In our current implementation, all intervals are treated as flat, i.e., after projection, the interval gets the average depth of the two end points. This approximation is acceptable because our intervals are very thin (even for thick curves as they are tessellated into multiple thin intervals). This makes the performance improvement with respect to intersections less important for our purposes. Disregarding intersections, our resolve is still more efficient than the previously described algorithm as they first need to sort all intervals and then maintain an additional sorted list of active intervals during the sweep. In contrast, we only need a single data structure that is manipulated very efficiently.

5 GPU Implementation

Our GPU pipeline resembles a recent software sort-middle pipeline [18]. The major difference is that our pipeline is used for rendering thin curves efficiently using line samples, rather than rendering triangles using point samples. In this section, we describe the different stages, namely, setup, binning, rasterization and sample

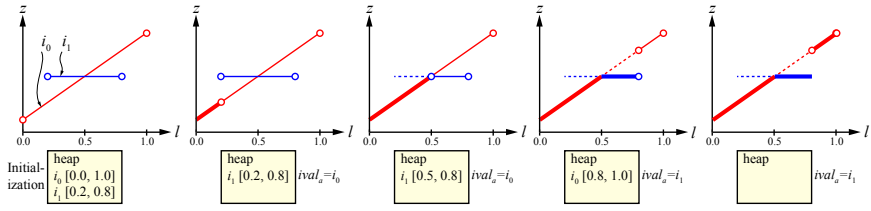


Figure 8: *Illustration of our resolve procedure (from left to right). We have two intervals, one red and one blue, and desire to find the closest interval segments along l. The fat lines indicate resolved color, which will be accumulated to the final pixel color. First, all intervals are ordered in a binary heap by their starting position along l (first diagram). The algorithms then works by sweeping l, keeping track of the interval currently being closest to the camera, that is, with the smallest z-value. At each encountered starting point, the closest interval is selected while the occluded interval is moved to the next-coming starting point (diagram 2). At intersections, the interval being occluded initially is clipped into a (potentially) visible part (diagram 3). This way, only two intervals needs to be processed at a time.*

blend, of the pipeline. The discussion in this section refers to pixels in the context of two line samples per pixel. A straightforward way to increase the number of line samples per pixel is to render the image in higher resolution, and then down sample the resulting image to the desired resolution. Our current implementation uses either two (no down sampling) or four (2× down sampling) axis-aligned line samples per pixel.

5.1 Setup

The input to the pipeline is a set of thin curves, defined by quadratic Bézier curves (BCs) with varying thickness (Equation 1). Our pipeline can either use interpolation of thin curves using the method in Appendix A, or directly render the curves without interpolation. For interpolation, we use thin *control* curves. Each thin control curve consists of a series of *control* Bézier curves (CBCs).

The purpose of this stage is to generate a record for each BC of each thin curve. A BC is identified by a 32-bit *BC identifier*, where the high 24 bits identify the thin curve index, and the low 8 bits identify the BC index. When interpolating a BC from the CBCs, the weights and base CBC indices are fetched using the thin curve index. To get the actual CBC indices, the base CBC indices are offset by the BC index. Each final record stores the projected bounding box as well as BC identifier. In the current implementation, each such record requires 16 bytes, where the last 4 bytes are used as padding to preserve alignment. If we instead opt to recompute the bounding box when needed, we can get away with as little as 4 bytes, which

Algorithm 1 Resolve

insert all intervals into *heap* ordered by *start*
$ival_a \leftarrow removeMin(heap)$
$l_s \leftarrow start(ival_a)$
while $heap \neq \emptyset$ **do**
  $ival_b \leftarrow removeMin(heap)$
  $l_e \leftarrow start(ival_b)$
  accumulate $ival_a \in [l_s, l_e]$ to current pixel
  $l_s \leftarrow l_e$
  if $end(ival_a) \leq l_s$ or $ival_a$ occluded by $ival_b$ after l_s **then**
    swap($ival_a, ival_b$)
  end if
  $i \leftarrow intersect(ival_a, ival_b)$
  if $i \neq nil$ and $i > l_s$ **then**
    clip part of $ival_b$ before i
    reinsert $ival_b$ in *heap*
  else if $end(ival_b) > end(ival_a)$ **then**
    clip part of $ival_b$ before $end(ival_a)$
    reinsert $ival_b$ in *heap*
  end if
end while
$l_e \leftarrow end(ival_a)$
accumulate $ival_a \in [l_s, l_e]$ to current pixel

might be preferred in cases where memory is scarce. In addition, each BC is culled against the view frustum. If a BC is culled, it is only flagged as such to maintain a fixed input-to-output mapping. The flagged BCs are removed in the next stage.

In order to handle thick curves, the width of a curve is estimated by calculating the projected radius of each control point. If this radius is larger than a pixel, the curve is flagged as thick. A parallel partition is used to make sure that all thick curves appear before any thin curves. This order is conserved throughout the pipeline.

5.2 Binning

This stage subdivides the screen into tiles of 128×128 pixels. The BCs are binned to per-tile lists, called *bin lists*. In order to avoid excessive synchronization, a large thread block is allocated per streaming multiprocessor (SM), so that only a single thread block is active in each SM at any given time. These thread blocks are kept running until all input data have been processed, similar to persistent threads [1]. We use thread blocks with 16 warps, i.e., 512 threads. For each tile, a separate bin list is kept for each SM, which allows for parallel insertion without inter-SM synchronization. The number of SMs on an NVIDIA GTX 580 GPU, for example, is 16, which makes this a reasonable approach. Each thread block reads chunks of

BCs, and may compact them due to culled segments. More chunks are read until we have 512 segments within the view frustum to process (or until all segments have been fetched). This ensures that all threads within the thread block have work to do. Shared memory is used to efficiently coordinate bin list insertion between threads [18].

5.3 Rasterization

This pipeline stage is divided into two separate kernels, *coarse* and *fine* rasterization. First, coarse rasterization is performed using the bin list of the tile, and then fine rasterization is performed to calculate the intersections with the line samples.

Coarse Rasterization Here, the BCs in each 128×128 tile is binned to the line samples within each tile. To determine whether a BC overlaps with a line sample, the BC's bounding box is simply tested against plane of the line sample.

Fine Rasterization During fine rasterization, each warp processes a single line sample. The line sample is divided into 32 4-pixel regions associated with memory for storing intervals overlapping that region. The current size of each list is kept in shared memory. This memory is allocated for each thread of a warp when the kernel is launched and is reused for each line sample processed by a warp. Typical memory requirements for these lists can be found in Section 6.

First, all BCs corresponding to thick curves are processed for the line sample. Each thread calculates the tessellated intersections for a single BC. The resulting intervals are shaded and appended to overlapping interval lists using shared memory atomics. Then, all remaining BCs are processed. Each thread calculates the intersection between the line sample and a BC. The resulting intervals are once again shaded and appended to overlapping interval lists using shared memory atomics. Once all BCs have been processed, each thread becomes responsible for a 4-pixel region and its associated interval list. Each interval list is then resolved according to the algorithm in Section 4.4 in parallel. Occlusion culling is performed both using the bounding box of each BC as well as for each interval before it is put into an interval list.

5.4 Sample Blend

After rasterization is complete, we essentially have two copies of the frame buffer; one for horizontal line samples and one for vertical line samples. The purpose of this stage is to combine the result of these sampling directions in a smart way. For each pixel, we blend between two line samples using a similar heuristic for curves that Jones and Perry [15] use for triangle edges. As an extension, the contribution by a curve is multiplied by its visible length along the line sample. The purpose of this extension is that misaligned occluded curves should not change how the

weighting is performed. Also, we attempt to punish misaligned curves by providing a *negative* weight for them. Below we give the details of this weighting for the reproducibility of our work.

If the curve tangent is projected to screen space, the angle, α, from the line sample to the tangent influences the weight. In particular, we use the weight:

$$w = \begin{cases} +\left(\frac{\sin\alpha - \sin\beta}{1-\sin\beta}\right)^2, & \text{if } \alpha > \beta \\ -\left(\frac{\sin\beta - \sin\alpha}{\sin\beta}\right)^2, & \text{otherwise} \end{cases}, \tag{6}$$

where β is the threshold were the curve is assigned a negative weight. We use $\beta = 15^\circ$. This weight is integrated during the interval resolve procedure along with the color of the interval. During sample blend, we have two weights, w_h and w_v, and two colors, $\mathbf{c}_h$ and $\mathbf{c}_v$, that represent horizontal and vertical line samples respectively. If any of the line samples have a negative weight, the sample with the largest weight will be picked. Otherwise, the following formula, using a smoothstep function, is used to blend between the samples:

$$\mathbf{c}_p = \mathbf{c}_h + (\mathbf{c}_v - \mathbf{c}_h) \cdot s^2(3-2s), \tag{7}$$

where $s = \frac{w_v}{w_h + w_v}$ and $\mathbf{c}_p$ is the final color of the pixel.

Sample blending is not perfect, e.g., when a single near horizontal curve is in front of multiple near vertical curves. In this case, both sampling directions may contain significant error. A simple way to deal with those cases is to use more than two line samples per pixel.

6 Results

Our implementation runs on an NVIDIA GTX 580, and all images in this paper were rendered at 1024×1024 pixels. For all our timings, we report how much time it took to render primary visibility and local shading. The shadows were baked into the control points of the Bézier curves using opacity shadow mapping [17] with line sampling. If n layers are used, we have seen that the shadow map generation takes approximately $n \cdot t$ ms, where t is the time for computing primary visibility. For simplicity, all our images were rendered with Kajiya and Kay's phenomenological appearance model for hair [16]. In the case of thick curves, we apply a simple normal variation that integrates to the same color as the corresponding thin curves. Our focus in this research is on accurate visibility, but we note that other appearance models, such as the one by Marschner et al. [21], should be possible to use as well. In all our images, we use two horizontal and two vertical line samples per pixel.

To make comparisons, we use two additional algorithms, namely, a modified version of Laine and Karras' CudaRaster [18] and OpenGL. CudaRaster is modified

HAIRY GUY	Zoom 1 (furthest)		Zoom 2		Zoom 3		...
Our	110 ms	*65.33* dB	130 ms	*60.83* dB	140 ms	*56.23* dB	
CudaRaster	1700 ms	56.62 dB	1150 ms	53.24 dB	660 ms	49.39 dB	
OpenGL	*20* ms	61.33 dB	*30* ms	58.51 dB	*90* ms	56.22 dB	

...	Zoom 4		Zoom 5		Zoom 6 (nearest)	
	140 ms	52.05 dB	220 ms	48.65 dB	250 ms	47.39 dB
	550 ms	45.40 dB	670 ms	42.05 dB	800 ms	40.38 dB
	80 ms	*54.75* dB	*140* ms	*52.26* dB	*160* ms	*51.13* dB

Table 1: *Rendering times for the teaser scene compared to other algorithms. Each of the six leftmost zoom-levels in the image are included here. Higher PSNR indicates closer resemblance to ground truth and is thus better. OpenGL is consistently faster, but overall only by a modest factor. At far distances, our algorithm achieves better image quality than OpenGL does. CudaRaster requires* $3-15\times$ *more rendering time than our algorithm, while also exhibiting lower PSNR.*

to support 64 samples per pixel and, since it does not support tessellation, the thin curves were tessellated in a separate stage and fed to the pipeline as triangles. In this separate stage, we tessellate until the screen-space area of the triangle formed by the control points of a curve is smaller than a threshold of 0.5 or 1, depending on the scene complexity. We take care to only tessellate curves within the view frustum in order to make the comparison as fair as possible. The time required for this tessellation stage have been excluded from all measurements of the performance of CudaRaster. The OpenGL implementation uses multi-sampling anti-aliasing (MSAA) and hardware tessellation with pre-tessellation frustum culling to achieve maximum performance. An MSAA rate of 8 was used in combination with down-sampling from a render target enlarged 4×4, for an effective rate of 128 samples per pixel. It should be noted that 128 is the maximum, single-pass sampling rate available for OpenGL, due to size restrictions of the render target. To reach a comparable curve quality, we implemented adaptive tessellation to sub-pixel size (0.25 pixel). We measure the quality of the algorithms by calculating peak-signal-to-noise (PSNR) figures between ground truth images and images produced by each of the algorithms. For ground truth, we use OpenGL with tiling to increase the sampling rate to 8192 samples per pixel.

In Figure 1, we show that our visibility algorithm can render practically noise-free visibility of a complex model from many different distances. The hair strands in this model are each unique, i.e., the thin control curve interpolation method in Appendix A was *not* used. As mentioned in the caption, the image to the left (where the camera is farthest away) renders in 110 ms, and the most detailed close-up of the face renders in 470 ms. Note that the leftmost image is relatively difficult to process quickly since there are many hair strands per pixel, which increases intersection computations. The rightmost image showcases our ability to render thick curves. Rendering times and PSNR figures for the different algorithms for this scene is shown in Table 1. It is clear that our algorithm outperforms CudaRaster in

Figure 9: *Some different types of fur rendered with our algorithm. From left to right, these models contain 50k, 100k, and 150k fur strands, where each fur strand consist of 8, 16, and 16 Bézier curves, respectively. The rendering times were (left to right) 78 ms, 358 ms, and 531 ms. Zoom in the pdf to explore the quality of the visibility.*

MINK			
Our	140 ms *56.24* dB	360 ms 41.37 dB	*130* ms 44.19 dB
CudaRaster	1420 ms 48.70 dB	1200 ms 34.91 dB	690 ms 38.58 dB
OpenGL	*60* ms 53.49 dB	*230* ms *43.30* dB	140 ms *50.63* dB

Table 2: *Rendering time comparisons for the middle ball of Figure 9 at various distances. Up close (right), our algorithm is able to cull large parts of occluded geometry and performs faster than OpenGL, even though PSNR is lower. At a distance (left), however, while being slower than OpenGL, our rendering has better PSNR.*

every case, both concerning PSNR and rendering time. OpenGL is usually better than our algorithm, but it is worth noting that we gain performance as we zoom in to the image. Our quality is also better when looking at the model from far away. It should be noted that when the model is far away, PSNR increases by default due to the presence of more white pixels.

In Figure 9 and 10, we show some renderings of different fur with different shader parameters with many fur strands per object. These models use the interpolation technique from Appendix A with 5,762 thin control curves. The fur ball to the right can be seen as a stress test for our algorithm, since it consists of $16 \cdot 150\text{k} = 2{,}400\text{k}$ Bézier curves. We show a detailed performance comparison of the ball in the middle in Table 2 at different zoom levels. From far away, we have better PSNR

Figure 10: *Closer rendering of the fur shown to the right in Figure 9.*

than OpenGL, but OpenGL is faster. In the close up, OpenGL is slower, but have better PSNR. CudaRaster performs worse than the other algorithms in all cases.

It is clear that OpenGL outperforms our algorithm in almost every scene. We are, however, very close and would like to stress that it is quite remarkable that a software-only approach can be this close performance-wise to a pipeline accelerated by fixed-function tessellation & rasterization units, and by color & depth compression units (which our algorithm cannot use, nor can CudaRaster). If the same algorithms were implemented on a CPU, we argue that the performance would be more comparable to the comparison between our algorithm and CudaRaster. Still, the fact that our algorithm runs extremely well on a GPU shows that it lends itself well to massive parallelization.

We investigated how our algorithm performs with respect to image resolution. For the fur ball to the right in Figure 9, we obtained the following rendering times:

512×512	768×768	1024×1024
328 ms	422 ms	531 ms

Figure 11: *Our algorithm renders the* GRASS *scene, containing 150k straws with a total of 3 million curve segments, in 406 ms.*

As can be seen, the time per pixel decreases with higher resolutions. In fact, the time per pixel for 1024×1024 is less than half of that for 512×512, for example. This is to be expected since there will be fewer intervals per pixel the higher resolutions we use, and also better parallel utilization of the GPU. The rendering time of a frame is typically divided into about 2.5% for coarse rasterization and 97% for fine rasterization, while the sum of the other stages (setup, binning, sample blend) is negligible. Hence, it is clear that it is the fine rasterization stage that should be optimized for improved performance. We leave this for future work.

Below we show the memory usage of our pipeline when rendering the fur ball in the middle in Figure 9:

BC records	Bin lists	Tile lists	Interval lists	Total
25.9 MB	9.7 MB	150.7 MB	38.4 MB	225MB

Here, *BC records* are the records created in the setup stage, *bin lists* represents the memory used for binning, *tile lists* represents the memory used for coarse rasterization, and *interval lists* represents the memory used for storing intervals during fine rasterization.

Another test scene rendered with our algorithm is shown in Figure 11.

7 Conclusions and Future Work

We have presented a novel visibility engine for thin curve rendering. Contrary to previous work, our method uses a single, continuous geometric representation from all view directions and at all scales. Our new resolve procedure for intervals contributes significantly to high performance, while using our intersection algorithm between thin curves and line samples contributes to high-quality visibility. There are many avenues for future work. First, it would be interesting and challenging to extend our approach to handle motion blur, perhaps with a method similar to how motion blur was dealt with for line sampled triangles [10]. Using stochastic simplification [5] of the thin curves, so that fewer, but fatter curves can be rendered from far away also seems worthwhile to explore, since performance would increase. It is clear that hair, especially blond hair, is transparent, and we believe that the method by Gribel [11] for resolving for visibility could be used off the shelf for this. However, we would also like to research faster methods for transparent resolve. Finally, we will also continue working on extended shadow mapping techniques. If the intervals are compressed, it should be possible to use a line sampled shadow map.

Acknowledgements

Thanks to Luca Fascione and Sebastian Sylwan from Weta Digital for the hair model. We acknowledge the Swedish Research Council, and Tomas is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

Bibliography

[1] AILA, T., AND LAINE, S. Understanding the Efficiency of Ray Traversal on GPUs. In *High Performance Graphics* (2009), pp. 145–149.

[2] BERTAILS, F., KIM, T.-Y., CANI, M.-P., AND NEUMANN, U. Adaptive Wisp Tree: a Multiresolution Control Structure for Simulating Dynamic Clustering in Hair Motion. In *Symposium on Computer Animation* (2003), pp. 207–213.

[3] CATMULL, E. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[4] COBB, E. S. *Design of Sculptured Surfaces using the B-Spline Representation*. PhD thesis, University of Utah, 1984.

[5] COOK, R. L., HALSTEAD, J., PLANCK, M., AND RYU, D. Stochastic Simplification of Aggregate Detail. *ACM Transactions on Graphics 26*, 3 (July 2007), 79:1–79:8.

[6] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, third ed. MIT Press, 2009.

[7] ELBER, G., KWON LEE, I., AND SOO KIM, M. Comparing Offset Curve Approximation Methods. *IEEE Computer Graphics and Applications 17* (1997), 62–71.

[8] ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. Stochastic Transparency. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (August 2011), 1034–1047.

[9] FARIN, G. *Curves and Surfaces for CAGD—A Practical Guide*, 5th ed. Morgan-Kaufmann, 2002.

[10] GRIBEL, C. J., BARRINGER, R., AND AKENINE-MÖLLER, T. High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility. *ACM Transactions on Graphics 30*, 4 (August 2011), 54:1–54:11.

[11] GRIBEL, C. J., DOGGETT, M., AND AKENINE-MÖLLER, T. Analytical Motion Blur Rasterization with Compression. In *High-Performance Graphics* (2010), pp. 163–172.

[12] HADAP, S., CANI, M.-P., LIN, M., KIM, T.-Y., BERTAILS, F., MARSCHNER, S., WARD, K., AND KAČIĆ-ALESIĆ, Z. Strands and Hair: Modeling, Animation, and Rendering. In *ACM SIGGRAPH 2007 courses* (2007).

[13] HAIN, T. F., AHMAD, A. L., RACHERLA, S. V. R., AND LANGAN, D. D. Fast, Precise Flattening of Cubic Bézier Path and Offset Curves. *Computers & Graphics 29*, 5 (2005), 656–666.

[14] HERY, C., AND RAMAMOORTHI, R. Importance Sampling of Reflection from Hair Fibers. *Journal of Computer Graphics Techniques 1*, 1 (2012), 1–17.

[15] JONES, T. R., AND PERRY, R. N. Antialiasing with Line Samples. In *Eurographics Workshop on Rendering* (2000), pp. 197–205.

[16] KAJIYA, J. T., AND KAY, T. L. Rendering Fur with Three Dimensional Textures. In *Computer Graphics (Proceedings of SIGGRAPH 89)* (1989), pp. 271–280.

[17] KIM, T.-Y., AND NEUMANN, U. Opacity Shadow Maps. In *Eurographics Workshop on Rendering Techniques* (2001), pp. 177–182.

[18] LAINE, S., AND KARRAS, T. High-Performance Software Rasterization on GPUs. In *High-Performance Graphics 2011* (2011), pp. 79–88.

[19] LEBLANC, A. M., TURNER, R., AND THALMANN, D. Rendering Hair using Pixel Blending and Shadow Buffers. *Journal of Visualization and Computer Animation 2*, 3 (1991), 92–97.

[20] LOKOVIC, T., AND VEACH, E. Deep Shadow Maps. In *Proceedings of ACM SIGGRAPH 2000* (2000), pp. 385–392.

[21] MARSCHNER, S. R., JENSEN, H. W., CAMMARANO, M., WORLEY, S., AND HANRAHAN, P. Light Scattering from Human Hair Fibers. *ACM Transactions on Graphics 22*, 3 (July 2003), 780–791.

[22] MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 23–32.

[23] MOON, J. T., AND MARSCHNER, S. R. Simulating Multiple Scattering in Hair using a Photon Mapping Approach. *ACM Transactions on Graphics 25*, 3 (July 2006), 1067–1074.

[24] MOON, J. T., WALTER, B., AND MARSCHNER, S. Efficient Multiple Scattering in Hair using Spherical Harmonics. *ACM Transactions on Graphics 27*, 3 (August 2008), 31:1–31:7.

[25] NGUYEN, H., AND DONNELLY, W. Hair Animation and Rendering in the Nalu Demo. In *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison Wesley, 2005, ch. 23, pp. 361–380.

[26] RUF, E. An Inexpensive Bounding Representation for Offsets of Quadratic Curves. In *High Performance Graphics* (2011), pp. 143–150.

[27] SINTORN, E., AND ASSARSSON, U. Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering. In *Symposium on Interactive 3D Graphics and Games* (2008), pp. 157–162.

[28] TILLER, W., AND HANSON, E. Offsets of Two-Dimensional Profiles. *IEEE Computer Graphics and Applications 4* (1984), 36–46.

[29] TZENG, S., PATNEY, A., DAVIDSON, A., EBEIDA, M. S., MITCHELL, S. A., AND OWENS, J. D. High-Quality Parallel Depth-of-Field Using Line Samples. In *High Performance Graphics* (June 2012), pp. 23–31.

[30] VAN WIJK, J. J. Ray Tracing Objects Defined by Sweeping a Sphere. *Computers & Graphics 9*, 3 (1985), 283–290.

[31] ZINKE, A. *Photo-Realistic Rendering of Fiber Assemblies*. Dissertation, Universität Bonn, 2008.

[32] ZINKE, A., AND WEBER, A. Light scattering from filaments. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (2007), 342–356.

[33] ZINKE, A., YUKSEL, C., WEBER, A., AND KEYSER, J. Dual Scattering Approximation for Fast Multiple Scattering in Hair. *ACM Transactions on Graphics 27*, 3 (2008), 1–10.

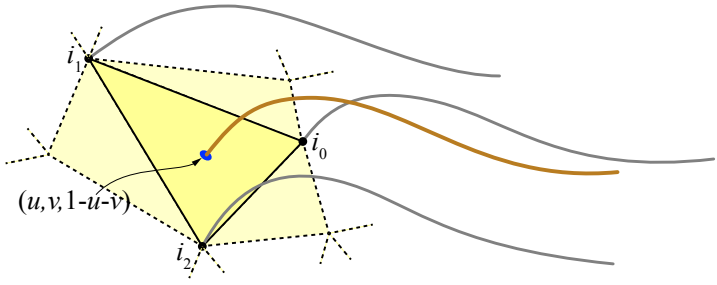


Figure 12: *Part of a triangular control mesh is shown where the grey curves are thin control curves defined at the vertices of a darker yellow triangle. Given the thin control curves, a new thin curve (brown) can be represented by thin control curve indices, (i_0, i_1, i_2), and barycentric coordinates, $(u, v, 1-u-v)$ within that triangle. This is an extremely compact representation which is valuable when rendering large numbers of thin curves.*

A Compact Representation

Here, we present a method that makes thin curve representation more memory efficient. To do this, we define thin curves by interpolating *thin control curves* that emanate from the vertices of a triangular control mesh. This means that there will be a thin control curve defined at each vertex of the triangle, and thin curves generated "over" the triangle using interpolation of the three thin control curves. An interpolated thin curve is represented by three thin control curve indices, (i_0, i_1, i_2), and barycentric coordinates, which are used to compute the starting point inside the triangle and to blend the thin control curves. See Figure 12. This is similar to the approach taken by Nguyen and Donnelly [25], with the exception that we keep this efficient representation throughout the rendering pipeline as well, rather than only using it for simulation. Conceptually, the triangular control mesh can be thought of as the scalp when performing hair rendering, and the landscape mesh when performing grass rendering, for example. With this approach, geometry is amplified by interpolating the thin control curves, which causes nearby curves to clump and align with each other. This behavior can be observed in, e.g., hair [2]. In practice, interpolated thin curves are generated by distributing point samples over the triangular control mesh. It should be noted that this compact representation is not a requirement for our visibility algorithm. Instead each, e.g., hair strand, can be defined as a complete thin curve without interpolation. In Section 6, both methods are used.

Paper III

Dynamic Stackless Binary Tree Traversal

Rasmus Barringer Tomas Akenine-Möller

Lund University

ABSTRACT

A fundamental part of many computer algorithms involves traversing a binary tree. One notable example is traversing a space partitioning acceleration structure when computing ray traced images. Traditionally, the traversal requires a stack to be temporarily stored for each ray, which results in both additional storage and memory bandwidth usage. We present a novel algorithm for traversing a binary tree that does not require a stack and, unlike previous approaches, works with dynamic descent direction without restarting. Our algorithm will visit exactly the same sequence of nodes as a stack-based counterpart with extremely low computational overhead. No additional memory accesses are made for implicit binary trees. For sparse trees, parent links are used to backtrack the shortest path. We evaluate our algorithm using a ray tracer with a bounding volume hierarchy for which source code is supplied.

Journal of Computer Graphics Techniques, 2(1):38–49, 2013.

Figure 1: CHESS *scene rendered using our stackless binary tree traversal algorithm.*

1 Introduction

Traversing a binary tree is a fundamental operation for many computer algorithms. One notable example, related to computer graphics, is traversing a space partitioning acceleration structure when performing ray tracing [5]. The traversal usually requires a stack to be temporarily stored that contains nodes that are still to be processed. However, in some cases, a stack is prohibitively expensive to maintain or access [4], e.g., for highly parallel architectures with many active traversal states. There may also be situations where the traversal state is suspended and resumed [1], in which case storing or transferring the full stack is expensive. For these reasons, stackless algorithms have been explored. Hughes and Lim [2] demonstrate stackless traversal for dense implicit kd-trees. Their approach requires a k-by-3 matrix to be stored in constant memory, where k is the depth of the tree. When traversal ascends in the tree, additional heuristics are needed to know which child to continue traversing, requiring additional knowledge of the data structure. Another approach that is not restricted to kd-trees involves using a short stack and encoding a restart trail in a bit mask [4]. When the short stack is insufficient, traversal restarts from the root node and descends along the stored restart trail. Hapala et al. [1] use parent pointers to achieve stackless traversal by backtracking. Their algorithm needs to determine traversal order among two siblings again when ascending in the tree. This is prohibitively expensive for anything but a simple ordering heuristic. As a result, the traversal order in their bounding volume hierarchy is based solely on ray direction. Computing the actual distance to the siblings' bounding boxes, and sorting them based on distance, would require re-intersecting both nodes to determine the traversal order.

In this paper, we introduce low overhead stackless traversal algorithms for binary trees that, unlike previous approaches, support dynamic descent direction without restarting. In particular, we introduce two algorithm variations for implicit binary trees, and one variation for sparse trees. Our algorithms will visit exactly the same sequence of nodes as a stack-based counterpart with extremely low computational

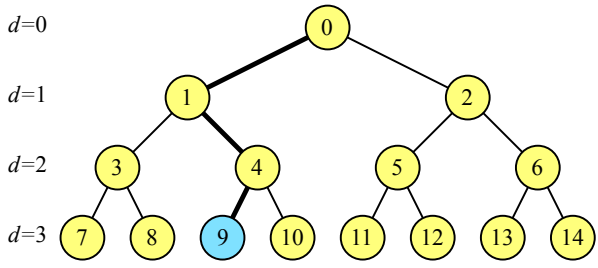


Figure 2: *An example of a complete binary tree with the depths, d, of each level shown to the left. The node number is shown inside each node circle. For the blue node at the bottom, we have* $levelStart = 2^d = 2^3$ *and* $levelIndex = 2$*, which makes it possible to compute the node number as* $2^3 + 2 - 1 = 9$*. For the blue node, we also have* $swapMask = 010$ *(in binary), which indicates the path from the root down to the node, where* 0 *is a descension in to the left child, and* 1 *is in to the right child. The swapMask is used in Algorithm 3.*

overhead. No additional memory accesses are made for implicit binary trees. For sparse trees, parent links are used to backtrack the shortest path.

2 Implicit Traversal Algorithm

For implicit binary trees, we store the nodes in each level sequentially in memory, i.e., the root node at location 0 and its left and right children at location 1 and 2, respectively. In general, the nodes at a depth, d, are enumerated as $\{2^d - 1, \ldots, 2^{d+1} - 2\}$, where $d = 0$ indicates the root level. As such, the relationships between different nodes is explicitly known. Given the address of a node, it is possible to calculate the address of the parent, children, and sibling.

The trees for, e.g., bounding volume hierarchies are generally not perfectly balanced. In those cases, we simply leave gaps in the memory layout for unused nodes. They will never be accessed so it is enough to allocate the address space for them. CPUs can utilize virtual memory to reserve the entire address space for the tree but only map pages that actually contain nodes. This reduces the memory overhead associated with filling out the gaps. In Section 3 it is shown that the algorithm can be extended to sparse binary trees with parent pointers.

We start by describing a simplified traversal algorithm that assumes that the left child in a tree is always traversed first, i.e., a typical depth-first traversal order. Then, we extend this algorithm to support tree traversal in any order.

2.1 Left-first Traversal

Knowing that we always choose the left child during traversal, we can keep track of the traversal state using two integers. We need one bit for each level of the tree and 32 bits is thus enough for reasonably balanced trees. However, for the purpose of our algorithm, integers of any size can be used. The algorithm is shown in Algorithm 2, where the first integer, *levelStart*, stores 2^d, where d is the current depth of the traversal. This variable is used to calculate the address of the first node in the current level, which is given by $levelStart - 1$. The second integer, *levelIndex*, stores the current node relative to the first node of the current depth level. The index of a node is thus given by $levelStart + levelIndex - 1$. This is illustrated in Figure 2.

Algorithm 2 Left-first Implicit Traversal

```
levelStart ← 1
levelIndex ← 0
repeat
    node ← levelStart + levelIndex − 1
    if node is leaf then
        process leaf
    else
        test node
        if accepted then
            levelStart ← levelStart ≪ 1
            levelIndex ← levelIndex ≪ 1
            continue
        end if
    end if
    levelIndex ← levelIndex + 1
    up ← ctz(levelIndex)
    levelStart ← levelStart ≫ up
    levelIndex ← levelIndex ≫ up
until levelStart ≤ 1
```

When the algorithm descends into the left child of a node, we recalculate the two integers to point to the left child in the next depth level. This is accomplished using simple shift operations.

If we decide to skip a node, or have found and processed a leaf node, we either need to traverse into the right sibling, or ascend upward in the tree. This is where we would normally need a stack containing the next node to process. However, since we always traverse into the left child first, we simply need to ascend in the tree when the right child of the parent node has been processed. This is equivalent to the criteria that *levelIndex* needs to have the same number of trailing zeros, as the number of levels to ascend in the tree. This approach is similar to what Knoll et al. [3] used for finding the leftmost root of implicit functions. The main difference is how they used intervals and a loop with floating-point division to

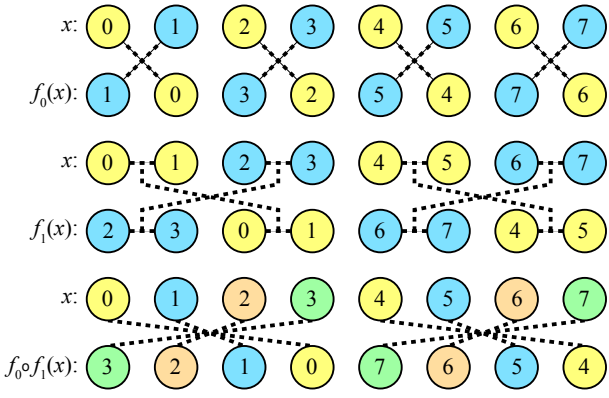


Figure 3: *The result of applying various combinations of $f_n(x)$ to a list of indices, x. Top: f_0. Middle: f_1. Bottom: $f_0 \circ f_1$.*

achieve the stackless traversal. Our approach can be implemented using a single count-trailing-zeros instruction (CTZ), which is common in many architectures. The integers are then adjusted by this amount using right shifts.

2.2 Generalized Traversal

When traversing a binary tree, it is often beneficial to start with a certain child based on a dynamic heuristic. For example, for efficient visibility computations using ray tracing, it is important to traverse into the child node whose content is most likely to shorten the ray, making traversal into the sibling unnecessary. A common heuristic is to start with the bounding volume that is closest. The traversal technique in Section 2.1, however, always traverses into the left child first, and if used in a ray tracer, performance would suffer substantially since rays would not benefit from early occlusion.

One way to enable arbitrary descension order (left or right child) is to actually traverse exactly as in Section 2.1 with the exception that the left and right children are swapped when traversal into the right child is preferred. It is, of course, not feasible to actually swap the memory of the nodes in the tree.

Algorithm 3 Generalized Implicit Traversal

```
levelStart ← 1
levelIndex ← 0
swapMask ← 0
repeat
    node ← levelStart + levelIndex − 1
          + swapMask − 2(levelIndex ∧ swapMask)
    if node is leaf then
        process leaf
    else
        test children of node
        if any accepted then
            levelStart ← levelStart ≪ 1
            levelIndex ← levelIndex ≪ 1
            swapMask ← swapMask ≪ 1
            if right child first then
                swapMask ← swapMask ∪ 1 {bitwise OR}
            end if
            if rejected one child then
                levelIndex ← levelIndex + 1
                swapMask ← swapMask ⊕ 1 {bitwise XOR}
            end if
            continue
        end if
    end if
    levelIndex ← levelIndex + 1
    up ← ctz(levelIndex)
    levelStart ← levelStart ≫ up
    levelIndex ← levelIndex ≫ up
    swapMask ← swapMask ≫ up
until levelStart ≤ 1
```

Instead, we define a function $f_n(x)$ that swap the indices of each node at a certain level in the tree:

$$f_n(x) = x + 2^n - 2(x \wedge 2^n), \tag{1}$$

where $\wedge$ represents bitwise AND. If the previous level took a right turn, the relative index would not be *levelIndex*, but rather $f_0(levelIndex)$. Some examples of applying this function to a list of indices is shown in Figure 3.

Since any level can take a right turn, we need to be able to apply composite functions to describe an arbitrary path through the tree:

$$(f_i \circ f_j \circ ...)(x), \tag{2}$$

where $i, j, \ldots$ are the levels where the traversal took a right turn and $(g \circ f)(x)$ is the same as $g(f(x))$.

Algorithm 4 Optimized Generalized Implicit Traversal

```
levelStart ← 1
levelIndex ← 0
levelIndex_dynamic ← 0
repeat
  node ← levelStart + levelIndex_dynamic − 1
  if node is leaf then
    process leaf
  else
    test children of node
    if any accepted then
      levelStart ← levelStart ≪ 1
      levelIndex ← levelIndex ≪ 1
      levelIndex_dynamic ← levelIndex_dynamic ≪ 1
      if right child first then
        levelIndex_dynamic ← levelIndex_dynamic + 1
      end if
      if rejected one child then
        levelIndex ← levelIndex + 1
      end if
      continue
    end if
  end if
  levelIndex ← levelIndex + 1
  up ← ctz(levelIndex)
  levelStart ← levelStart ≫ up
  levelIndex ← levelIndex ≫ up
  levelIndex_dynamic ← levelIndex_dynamic ≫ up
  levelIndex_dynamic ← levelIndex_dynamic + 1 − 2(levelIndex_dynamic ∧ 1)
until levelStart ≤ 1
```

From Figure 3, one can see that the result is independent of the order of composition, i.e., $f_i \circ f_j(x) = f_j \circ f_i(x)$, which suggests that the entire composite function can be expressed in a very simple manner. Let d denote the current depth of the traversal. If we create a bitmask, called *swapMask*, where each bit i is set if level $d-i-1$ took a right turn, then the entire composition becomes:

$$f_{\text{comp}}(x) = x + swapMask - 2(x \wedge swapMask). \tag{3}$$

Given $f_{\text{comp}}(x)$, we are now ready to describe the generalized traversal algorithm. We need one more integer to store *swapMask*. Besides some simple bookkeeping for *swapMask*, all we need to do is apply Equation 3 to *levelIndex*. The generalized traversal technique is shown in Algorithm 3.

In this algorithm, we essentially compute the dynamic descent *levelIndex* from a left-first *levelIndex* at each iteration. It is also possible to incrementally update the dynamic *levelIndex*. Denote the dynamic *levelIndex* as $levelIndex_{dynamic}$. It is

obvious that any dynamic descent can incrementally update $levelIndex_{dynamic}$ by assigning it the child traversed. We also observe that any ascension in the tree is independent of descent order; the parent at a given level is the same whether we traversed the left or the right child. The problem becomes which child to continue traversing after ascension. This is easy to answer since $levelIndex_{dynamic}$ indicates which child has already been traversed. We simply need to switch to the sibling of $levelIndex_{dynamic}$ after ascending in the tree. These observations are summarized in Algorithm 4. The main difference compared to Algorithm 3 is that the swap only occurs at the current level after ascending in the tree.

Even though Algorithm 3 indicates separate variables for $levelStart$ and $levelIndex$, it is possible to combine them into one. The start of the current level, $levelStart$, is always represented by a single set bit that is higher than any set bit in $levelIndex$. By introducing $index = levelStart + levelIndex$, we can replace all instances of the variables with $index$. This works because all operations on $levelIndex$ are unaffected by the high bit from $levelStart$. This optimization reduces the number of state variables to two, and avoids one addition and two redundant shift operations. The same optimization can be introduced in Algorithm 4 by setting $index = levelStart + levelIndex_{dynamic}$.

3 Sparse Traversal Algorithm

Sometimes the restrictions of an implicit binary tree cannot be met, e.g., when the tree is badly balanced or when the implicit memory layout cannot be used. In those cases, a stackless algorithm can still be used [4, 1]. It turns out that Algorithm 4 is actually well suited for sparse trees as well, given that there is a method to ascend in the tree. Assuming the existence of parent pointers that allows us to backtrack [1], we replace both $levelStart$ and $levelIndex_{dynamic}$ with a single node pointer. The node pointer is updated analogously to $levelIndex_{dynamic}$. When descending in the tree, the node pointer simply follows the appropriate child link. When ascending in the tree, instead of jumping to the appropriate parent using a shift instruction, we backtrack using the parent pointers until the destination level is reached. The swap function at the end of the loop is replaced by an analogous function that determines the sibling of a node. It can either be implemented by calculating the sibling directly, or, by taking a round trip to the parent, depending on the memory layout of the tree. The algorithm for sparse trees is shown in Algorithm 5.

4 Results

Our three algorithms, referred to as IMPLICIT-A (Algorithm 3), IMPLICIT-B (Algorithm 4) and SPARSE (Algorithm 5), were implemented in a simple single-threaded CPU ray tracer written in C++. All tests were performed on an Intel Core i7 pro-

Algorithm 5 Sparse Traversal

```
levelIndex ← 0
node ← root
repeat
  if node is leaf then
    process leaf
  else
    test children of node
    if any accepted then
      levelIndex ← levelIndex ≪ 1
      node ← left or right child
      if rejected one child then
        levelIndex ← levelIndex + 1
      end if
      continue
    end if
  end if
  levelIndex ← levelIndex + 1
  while levelIndex ∧ 1 = 0 do
    node ← parent(node)
    levelIndex = levelIndex ≫ 1
  end while
  node ← sibling(node)
until node = root
```

cessor clocked at 2.66 GHz and with 8 GB of 1067 MHz DDR3 RAM. The target compiler was Apple Darwin LLVM GCC 4.2.1. For reference, a conventional stack based traversal, STACK, was included. For comparison to other stackless approaches, we implemented the algorithms by Laine [4] and Hapala et al. [1], referred to as LAINE and HAPALA respectively. For LAINE, we investigate different short stack sizes and denote them LAINE-x, where x indicates the size of the short stack. The ray tracer uses a bounding volume hierarchy (BVH), containing axis-aligned bounding boxes (AABBs), optimized using surface area heuristic (SAH) for the first 8 depth levels, and median split along the longest axis of separation for the rest in order to get a reasonably balanced tree. The different traversal algorithms use exactly the same BVH. The implicit traversal algorithms, IMPLICIT-A/B, will, however, move all nodes to their predetermined memory location, including leaving gaps for unused nodes. Note that while IMPLICIT-A/B and SPARSE will visit exactly the same sequence of nodes as STACK, HAPALA will not due to its need for a simple traversal order heuristic. While the other algorithms will start with the closest child node, HAPALA will base its order on the ray direction along the split axis of a node. Although LAINE-x uses the same traversal order as STACK, it performs restarts from the root node. In the test setup, we used the CHESS scene shown in Figure 1 rendered at a resolution of 800×300 pixels, and the HAIRBALL scene shown in Figure 4 rendered at a resolution of 512×512 pixels. Both scenes use 16 eye rays and 256 ambient occlusion (AO) rays per pixel.

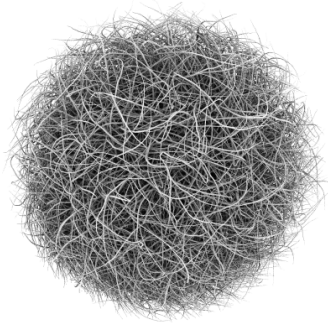


Figure 4: HAIRBALL *scene with* 2.8 *million triangles.*

	CHESS	HAIRBALL
STACK	24.6 s	149 s
SPARSE	26.5 s	157 s
IMPLICIT-A	27.5 s	164 s
IMPLICIT-B	26.2 s	160 s
HAPALA	35.8 s	189 s
LAINE-0	44.7 s	368 s
LAINE-1	31.6 s	232 s
LAINE-2	28.1 s	191 s
LAINE-4	26.4 s	166 s
LAINE-8	26.1 s	155 s

Table 1: *Table showing the time required for each algorithm to render the test scenes.*

The times required to render the scenes for the different algorithms are shown in Table 1. A more detailed performance break-down is shown in Figure 5.[1]

It is clear that our stackless algorithms are very competitive in terms of performance while maintaining a small traversal state. This is true for both the simpler CHESS scene, containing 64k triangles, as well as the more complex HAIRBALL scene containing 2.8M triangles. For the CHESS scene, IMPLICIT-B performs best of our algorithms and is only outperformed by a full stack (STACK) or a longer short stack (LAINE-8), while using a fraction of the memory for its traversal state. In HAIRBALL, SPARSE becomes faster than IMPLICIT-B, even though it backtracks using parent pointers. One explanation is the padding of the implicit BVH; the extra pages creates more cache misses in the translation lookaside buffer. IMPLICIT-B is consistently faster than IMPLICIT-A due to a lower traversal cost. LAINE-x gets worst performance for $x = 0$ in both test scenes due to frequent restarts. As the short stack increases in size, the performance approaches STACK at the cost of a larger traversal state. From Figure 5, it is clear that even LAINE-0 have larger traversal state than the other stackless algorithms. The performance of HAPALA is in the lower end of the performance spectrum in our tests.

5 Conclusion and Future Work

We have presented stackless traversal algorithms for both implicit binary trees and sparse binary trees with parent pointers. These algorithms use efficient bit manipulation and have low computational overhead. At the same time they support dynamic descent direction without having to re-evaluate sibling order or restarting. Our algorithm for sparse trees with parent pointers has been shown to perform well

[1] The intersection and traversal ratios have been measured by prohibiting the compiler from inlining the intersection tests, and sampling using a profiler (Instruments 4.5 Time Profiler). The ratios are thus based on a slightly different workload.

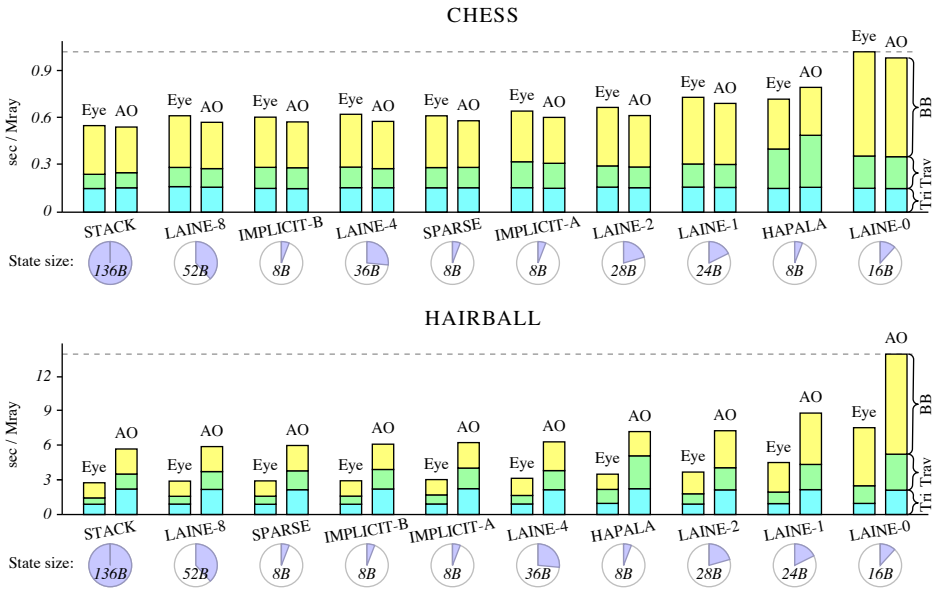


Figure 5: *Bar charts showing detailed performance measurements from* CHESS *(top) and* HAIRBALL *(bottom). Each algorithm shows the performance of eye and AO rays. Additionally, a time break-down into triangle intersection, traversal and AABB intersection is illustrated within each bar. Below each algorithm, the size of its traversal state is shown. The state size is based on the number of state variables needed (4 bytes each). A non-zero stack size adds an extra state variable for the stack head, and 4 bytes for each entry. The algorithms are ordered according to increasing rendering time. Note that the ordering is biased toward AO ray performance since they represent the majority of rays traced.*

while requiring only a minimal traversal state of two variables. Of our two algorithms for implicit trees, Algorithm 4 is most performant in our tests and should generally be preferred. A possible exception would be if the direction bit mask maintained by Algorithm 3 is useful for other purposes. In the future, we would like to investigate more uses for implicit binary trees, e.g., solving mathematical optimization problems where each node can be seen as an interval over an objective function. As a concrete example, our implicit traversal algorithm can be used to find the closest point on a curve evaluated using interval arithmetic. Another promising avenue for future work is investigating how stackless traversal algorithms can be employed by specialized ray tracing hardware. They can potentially be very useful for dynamic ray reordering during traversal in order to increase memory locality.

Acknowledgements

The authors thank the anonymous reviewers for their valuable comments and suggestions. The HAIRBALL model is courtesy of Samuli Laine. Tomas is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

Bibliography

[1] HAPALA, M., DAVIDOVIC, T., WALD, I., HAVRAN, V., AND SLUSALLEK, P. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011* (2011), pp. 29–34.

[2] HUGHES, D. M., AND LIM, I. S. Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1555–1562.

[3] KNOLL, A., HIJAZI, Y., KENSLER, A., SCHOTT, M., HANSEN, C., AND HAGEN, H. Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Computer Graphics Forum 28*, 1 (2009), 26–40.

[4] LAINE, S. Restart Trail for Stackless BVH Traversal. In *High Performance Graphics 2010* (2010), pp. 107–111.

[5] WHITTED, T. An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (1980), 343–349.

Paper IV

A^4: Asynchronous Adaptive Anti-Aliasing using Shared Memory

Rasmus Barringer Tomas Akenine-Möller

Lund University

ABSTRACT

Edge aliasing continues to be one of the most prominent problems in real-time graphics, e.g., in games. We present a novel algorithm that uses shared memory between the GPU and the CPU so that these two units can work in concert to solve the edge aliasing problem rapidly. Our system renders the scene as usual on the GPU with one sample per pixel. At the same time, our novel edge aliasing algorithm is executed asynchronously on the CPU. First, a sparse set of *important* pixels is created. This set may include pixels with geometric silhouette edges, discontinuities in the frame buffer, and pixels/polygons under user-guided artistic control. After that, the CPU runs our sparse rasterizer and fragment shader, which is parallel and SIMD:ified, and directly accesses shared resources (e.g., render targets created by the GPU). Our system can render a scene with shadow mapping with adaptive anti-aliasing with 16 samples per *important* pixel faster than the GPU with 8 samples per pixel using multi-sampling anti-aliasing. Since our system consists of an extensive code base, it will be released to the public for exploration and usage.

ACM Transactions on Graphics, 32(4):100:1–100:10, 2013.

1 Introduction

Geometric aliasing is still one of the major challenges in real-time rendering, as noted by Andersson [6] among others. Supersampling anti-aliasing (SSAA) is expensive both in terms of memory bandwidth usage, and in terms of fragment shading since each visibility sample is shaded individually. Multi-sampling anti-aliasing (MSAA) is less expensive, since the fragment shader is only executed once per pixel per primitive even though there are more visibility samples. Still, this incurs a lot of overhead in terms of rasterization, color & depth buffer memory bandwidth, and shading usually increases along triangle edges (see Section 2 for more information about this). At the same time, we note that many desktops and laptops have four or more CPU cores, and often, only a fraction of them are active during game play. A major goal of our research has been to develop an adaptive anti-aliasing (AA) algorithm where the CPU cores and GPU cores join forces to solve this problem using a shared memory architecture.

Already in 1977, Crow suggested to apply more expensive anti-aliasing techniques *only* to pixels being covered by the geometrical edges [13]. Algorithmic variants based on the same underlying idea have been proposed [25, 2] after that, but there is still no practically useful algorithm that also generates high image quality with high performance. Based on Crow's observation that only a sparse set of pixels needs high-quality edge anti-aliasing, we present a novel algorithm for solving the geometrical edge anti-aliasing problem.

Our algorithm leverages idle CPU cores to perform anti-aliasing for a sparse set of pixels, while allowing the GPU to render the entire scene quickly using a single sample per pixel (spp). There are several merits to this approach. First, since accurate anti-aliasing is calculated for limited parts of the frame buffer, the workload is control flow divergent. A current CPU is thus a better fit than current GPUs to perform such calculations. Second, since anti-aliasing is decoupled from the GPU rendering pipeline, we can anti-alias only the most important pixels, which is often less than 5% in our experience, and in the worst case early-out in order to guarantee a certain frame rate. Third, our target architecture exploits shared memory between the CPU and GPU, which makes it possible for the sparse fragment shader evaluation done on the CPU to directly (without copy) access render targets already generated by the GPU, and also sparsely update the final image with high-quality anti-aliased pixels. Together this makes our algorithm extremely fast. A high-level illustration of our algorithm can be seen in Figure 1.

2 Previous Work

There is a wealth of literature on the topic of post-process screen-space anti-aliasing. The first technique in this area is called morphological anti-aliasing (MLAA) [24], where the idea was to analyze the rendered image, detect edges, and cleverly filter over them with the goal of approximating an anti-aliased im-

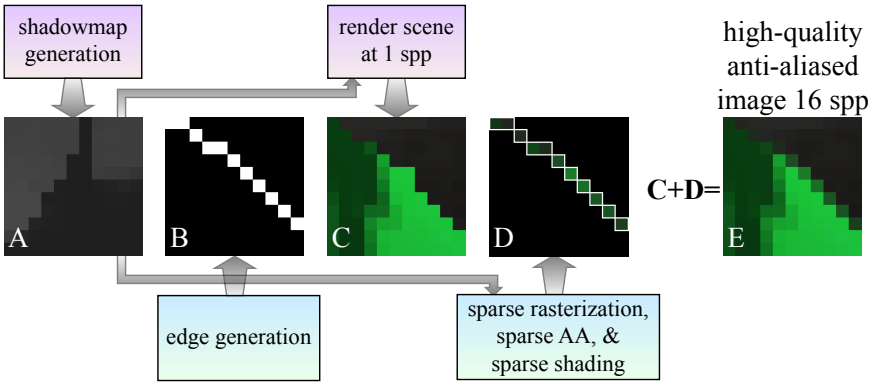


Figure 1: *Our system generates high-quality edge anti-aliasing. Two render targets are rendered (top) with one sample per pixel (spp)—a shadow map and a rendering of the scene using the shadow map. In addition, our algorithm generate edges (bottom) and perform sparse rasterization, anti-aliasing (AA), and shading (bottom) for the edge pixels in the scene (but not in the shadow map). What is unique about our system is that all the squares in the middle are located in shared memory, the passes at the top are executed by the GPU, and the passes at the bottom are executed on the CPU, and together they generate a high-quality anti-aliased image (right) with 16 spp for the edges.*

age. Since then, many intelligent edge-blur techniques have been proposed, and they are widely used in the game industry, since they are fast and power-efficient. We refer the interested reader to the excellent SIGGRAPH course on this topic by Jimenez et al. [20]. We note that these techniques are not robust to all kinds of scenarios.

Point sampling techniques, such as SSAA and MSAA (mentioned in Section 1), are commodity features in graphics architectures today. One potential disadvantage of MSAA is that shading along edges increases. For example, if a pixel is intersected by an edge (does not need to be a silhouette) shared by two triangles, and if there is at least one sample inside each triangle, then shading will be executed twice, for such pixels, and it will be done on a per 2×2 pixel basis. This can be particularly expensive for highly tessellated geometrical objects. Fatahalian et al. [14] present a hardware-based approach to solve this, where fragments are gathered and merged in order to avoid unnecessary shading. In coverage-sampled AA (CSAA) [27], each pixel stores at most 2^n colors in a per-pixel palette, and each sample may point into this palette using an n-bit index. For high-quality visibility, the number of coverage samples per pixel is higher than the number of colors in the palette, e.g., a pixel may have 4 colors and 16 coverage samples. When more than 4 colors appear in a pixel, a heuristic is needed to reduce that set of colors down to 4 colors again. Note that these techniques, including SSAA, MSAA, and

CSAA, are still rather brute-force approaches, since anti-aliasing is not directed only towards the pixels that are in need of it.

Carpenter's A-buffer [10] is another type of MSAA, where shading is decoupled from color and depth samples. Each rendered polygon stores a fragment per pixel, where a fragment consists of a coverage mask, i.e., a bitmask, as well as a color and some depth representation. A pixel can store any number of fragments in order to capture transparency effects etc. In addition, merging of fragments can be done in some situations for more efficient processing.

Crow [13] suggested that more effort should be spent on edges, and in particular on geometrical edges. For such pixels, analytical methods would be used to compute the area of coverage, and accumulated to the pixel color. However, this assumed non-overlapping polygons, and hence no handling of depth, which makes its usage less influential. Based on Crow's observation, however, Sander et al. [25] rendered the scene using 1 spp, and then overdrew the discontinuity edges with anti-aliased lines. This requires the lines to be rendered in back-to-front order for correct blending, and depth may not be resolved correctly on these lines, since the depth buffer is not multi-sampled. Aila et alia presented a hardware mechanism called the delay stream [2]. One of their applications was adaptive AA on geometrical edges, also based on Crow's observation. However, the delay stream is not yet implemented in any hardware to our knowledge. Our approach is also based on spending more AA efforts on geometrical edges, but our approach is based on introducing a sparse rasterizer & shader, and our method exploits shared memory to efficiently implement complex shading.

Greene et al. presented hierarhical Z-buffering [18], where a quad tree of depths is maintained, and each node stores the maximum depths of its childrens' depths. An object-space octree of the scene geometry is used for hierarchical culling against the Z-pyramid. Greene and Kass [17] extended the previous method to render scenes with guaranteed error bounds. An octree of the geometry is traversed in front to back order, and occluded geometry is culled against the quadtree. Potentially visible polygons are inserted into the quadtree that contains conservative Z_{min} and Z_{max} of the geometry in the node. Finally, the hierarchical tiling algorithm for high-quality rasterization by Greene [16] uses a coverage hierarchy. Classification of the polygon during traversal of the hierarchy is divided into inside, outside, or intersecting the nodes. Similarly, a quadtree node may be marked as fully covered, vacant, or active. Polygons are traversed in a strict front-to-back order using a BSP-tree with a Warnock-style [26] tiler for rasterization.

3 Algorithm Overview

Our algorithm is divided into two parts, where one is executed on the graphics processing unit (GPU), and the other on the CPU cores. The idea is to take any application, e.g., a game, and render the entire scene, as usual on the GPU, using either OpenGL or DirectX, at one sample per pixel (spp). The fragment shaders

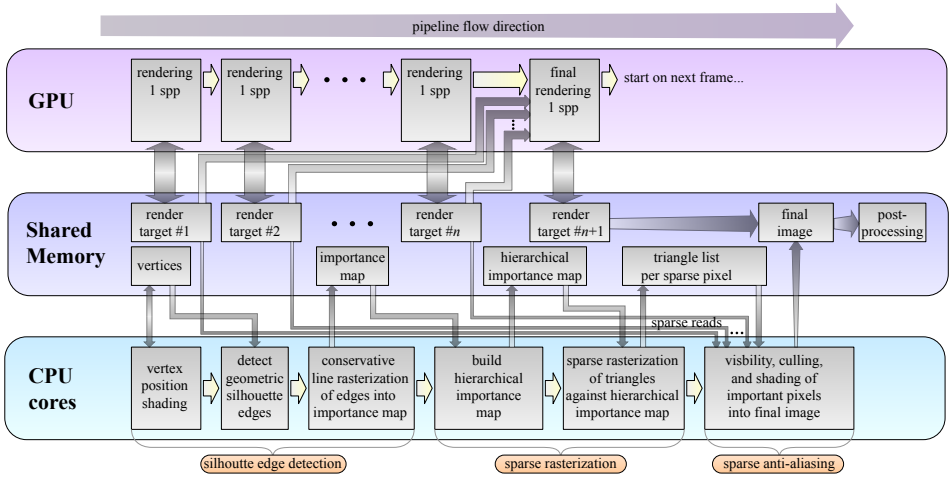


Figure 2: *In this example, we assume that n render targets, with one sample per pixel (spp), are rendered by the GPU, and that for the final rendering on the GPU, all render targets are read and used in the fragment shader for the final image. One or more post-processing passes may follow. The work of the GPU is shown at the top, the work done by the CPU cores is shown at the bottom, and the shared memory, which glues together the work done by the CPU and the GPU, is shown in the middle. Note that the CPU rendering pipeline consists of three stages, namely, silhouette edge detection, sparse rasterization, and sparse anti-aliasing. The CPU pipeline works on a sparse set of important pixels (see Figure 1) that are in need of high-quality anti-aliasing. One reason that our algorithm is fast is that it can exploit the render targets created rapidly by the GPU, and sparsely read from these render targets directly from the CPU, thereby avoiding the often expensive copy from GPU to CPU. In the same way, the CPU can update a sparse set of pixels directly into the final image since the memory is shared.*

used for rendering can include all conventional rendering techniques, such as local illumination, texture mapping, environment mapping, screen-space ambient occlusion, G-buffer passes for deferred rendering, light accumulation, shadow mapping, shadow volumes, etc. Most rendering engines consists of several passes, where each pass may generate one or more render targets (RTs). The user of our algorithm can choose to apply high-quality anti-aliasing to any subset of the render targets, and we call these *high-quality render targets* (HQRTs). The high-level idea of our approach is to first render to the RTs and the HQRTs as usual using the GPU with one spp, and then let the CPU refine the pixels, in the HQRTs, that are in need of high-quality anti-aliasing.

While the GPU is no more occupied than usual, our novel contribution is to run our adaptive anti-aliasing algorithm asynchronously on the CPU cores for the HQRTs.

Our CPU rendering pipeline consists of three stages, namely, *i*) silhouette edge detection, *ii*) sparse rasterization, and *iii*) sparse anti-aliasing. The number of pixels containing silhouette edges in an image is relatively small (often less than 5% in our experience), and more expensive, higher-quality anti-aliasing will be applied to only this sparse set of *important* pixels. The fragment shading for the sparse set of pixels is done on the CPU, where the render targets are accessed via shared memory. This makes our CPU shading extremely efficient. Finally, the high-quality versions of the sparse set of important pixels are written back to the HQRT, also located in shared memory.

In our experience, it is often sufficient to let only the final render pass (before any post-processing techniques, such as tone mapping, etc, are applied) render to an HQRT and let all other render targets be rendered at one spp on the GPU. This makes the impact of our algorithm very small. An illustration of such a setup is shown in Figure 2, where the final render pass uses an HQRT. In this setup, the fragment shader in the final render pass also accesses all previously generated render targets. Note that this is just an example—there are essentially an endless number of variations possible.

4 GPU Rendering Pipeline

The task of the GPU is simply to render all RTs and all HQRTs as usual with only 1 spp. As we will see in Section 5, the RTs and HQRTs may be accessed from the fragment shader running sparsely on the CPU, and in addition, the HQRTs will be updated sparsely with high-quality anti-aliased pixels. Hence, the RTs and HQRTs need to be shared between the CPU and the GPU. In the worst case, sharing resources means copying data from the GPU to the CPU every frame, and in the best case, it simply means reading from the same memory pointer as the GPU pipeline uses. The latter is a reality when working with a shared memory architecture, such as the Intel Ivy Bridge [22] with an integrated graphics processor. Therefore, our algorithm is designed around an architecture that can share the address space between the CPU and the GPU.

Even without shared memory, a duplication of static vertex buffers and textures generally works well. However, rendering gets substantially less efficient when there is a frequent use of render targets. We elaborate on this in Section 5.3.

5 CPU Rendering Pipeline

As mentioned in Section 3, the purpose of the CPU rendering pipeline is to detect a sparse set of *important* pixels that contain geometrical silhouette edges, and then sparsely apply a high-quality anti-aliasing algorithm to only these pixels. Our CPU pipeline is heavily optimized for rasterizing triangles to this sparse set of *important* pixels, and in this section, we describe the algorithmic side of our approach,

while the implementation details, which depends on the target architecture, are described in Section 6. The three stages of our CPU rendering pipeline are silhouette edge detection, sparse rasterization, and sparse anti-aliasing, and these stages are executed for all HQRTs. The following subsections contain descriptions of these stages.

Similar to the GPU pipeline, the input to our pipeline is organized into *draw calls*. Each draw call contains information about a group of vertices with the same format and rendering state, connected into triangles by an implicit relationship, or explicitly by a list of indices. The following subsections will refer to vertices and triangles in the context of all draw calls.

5.1 Silhouette Edge Detection

Here, we describe a straightforward silhouette edge detection mechanism, illustrated by the three leftmost gray boxes at the bottom in Figure 2, used in our algorithm. First, the clip-space positions for all vertices are computed. Note that we avoid computing other per-vertex attributes, and defer them until they are possibly needed. As will be seen later in Section 5.3, the vertex attributes are only computed sparsely for the vertices that are accessed. Next, the triangle edges are tested to determine whether they are geometrical silhouette edges. The usual convention is to treat an edge as a silhouette if the edge is shared by one front-facing and one back-facing triangle for closed models, or if the triangle edge is only connected to a single front-facing triangle [12, 9]. As a side effect, we mark edges as silhouettes whenever there is a difference in attributes on the edge, since the vertices are then distinct in the vertex buffer. For example, all edges of the quadrilaterals of a cube will be marked as silhouettes because the normals of the quadrilaterals differ.

All silhouette edges are then clipped against the canonical clip-space volume. The silhouette edges, which possibly have been clipped, are then conservatively rasterized into an image, called the *importance map*, with the same resolution as the HQRT. Note that we use the term *important* to indicate that a more sophisticated anti-aliasing algorithm should be applied to those pixels. Hence, it suffices with a single bit per pixel in the importance map, where zero indicates no further need of anti-aliasing. For conservative line rasterization, we use the two-dimensional version of Amanatides and Woo's DDA algorithm [5].

Although geometrical silhouette edges are unable to capture aliasing from intersecting triangles, a trivial extension can allow additional important pixels to be specified from, e.g., discontinuities in the color buffer, where there is no discontinuity in depth. Another possibility is to allow an artist to paint importance on objects and rasterize those to the importance map. Yet another is simply to mark an entire triangle or object as important. This may be useful when the sampling rate needs to be locally higher for an entire object. We refer to these additions to the importance map as *manual additions*. Manual additions represent a special case because some optimizations cannot be applied when performing shading culling, as described in Section 5.3. In particular, we cannot assume that there are

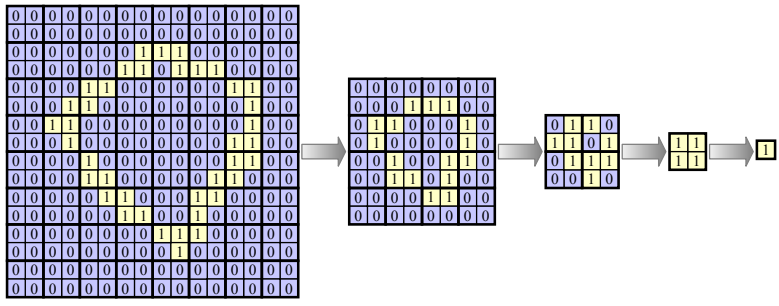


Figure 3: *The input from the edge detection phase is an importance map (left), with one bit per pixel indicating whether there is at least one silhouette edge overlapping the pixel. Here, the resolution is* 16×16*, but in reality, it may be* 2048×1024*, for example. Tree construction is done bottom up (illustrated as left to right in the figure). The bit of a parent is the logical* `OR` *of all children's bits.*

no intersecting triangles that are in need of AA.

5.2 Sparse Rasterization

This subsection describes how our sparse rasterizer, corresponding to the fourth and fifth gray boxes at the bottom in Figure 2, works. Early rejection of candidate triangles is one of the most important design principles of our CPU rasterizer, which makes this stage extremely important.

The sparse rasterization stage commences by creating a quadtree of the importance map from the edge detection phase (Section 5.1). Recall that the importance map contains one bit per pixel, and the bit is set to one if there is at least one silhouette edge overlapping the pixel. We call the resulting quadtree a *hierarchical importance map* (HIM), which is a full tree since the input is the entire image. The HIM will be used in the sparse rasterizer to stop traversal if a triangle does not overlap any pixels that are important. In our case, the bit of a parent is set to one if at least one children bit is one. An example is shown in Figure 3. Note that we have chosen a quadtree, but any hierarchical tree would work.

Next, we describe our sparse rasterization algorithm. The purpose is to find, for each pixel, a list of triangles that may possibly affect the final pixel color. Note that it does not suffice to loop over triangles with silhouette edges, because a triangle without a silhouette edge may still be visible within an important pixel, and can therefore affect the final color of the pixel. A typical example is a background triangle, and a silhouette edge cutting through a pixel. Therefore, *all* triangles are rasterized against the HIM in this stage.

In theory, sparse rasterization is simply the process where a triangle is traversed against the quadtree of the HIM. Traversal continues down into a child node if the

triangle overlaps with the spatial extents of the child node, and the corresponding bit in the HIM is set to one, i.e., there is at least one important pixel in the subtree of the child node. The overlap test between a quad and a triangle can be done with an efficient tile test using only additions [23, 3]. In the following, we will use the term *tile* to denote a $2^n \times 2^n$ region of pixels, where $n \geq 0$, and n can be adjusted for different performance trade-offs. When a node, whose size is equal to the tile size, is reached during traversal, a pointer to that triangle is added to the tile's triangle list so that the sparse anti-aliasing procedure (Section 5.3) knows which triangles to process in the important pixels. This means that a triangle list will be created for each tile, and these triangles are a superset of the triangles that may affect the final color of the pixels in that tile. In practice, the entire traversal can be done in a more efficient manner, depending on the target architecture. Our current implementation is described in Section 6.

Occlusion culling is very important to performance, and our approach is similar to z_{max}-culling used in graphics processors [21, 18], where conservative tests are used. For each tile, a z_{max}^{tile}-value is first initialized to ∞. If an opaque triangle is processed, and it covers the entire tile, the triangle's maximum depth, z_{max}^{tri}, inside the tile is computed as the maximum of the triangle depths at the four corners of the tile. If $z_{max}^{tri} < z_{max}^{tile}$ then the maximum depth of the tile is updated: $z_{max}^{tile} = z_{max}^{tri}$. Note that all triangles behind z_{max}^{tile} can be safely culled for opaque geometry. For all triangles that are processed during sparse rasterization, a z_{min}^{tri}-value is also computed as follows. The z_{min}^{verts} is computed as the minimum of the triangle vertices, and z_{min}^{corner} is computed as the minimum of the depths of the triangle plane at the four corners of the tile. A tight, conservative estimation of the minimum depth of the triangle in that tile is then $z_{min}^{tri} = \max(z_{min}^{verts}, z_{min}^{corner})$ [4]. This z_{min}^{tri}-value is stored with the triangle for that tile. All incoming triangles are culled against the current z_{max}^{tile}, i.e., if $z_{min}^{tri} > z_{max}^{tile}$ then the triangle is *not* added to the triangle list of the tile.

Note that our traversal and data structure (HIM) resemble the work of Greene et al. [18, 17, 16]. However, in our case, the quadtree is built once and used for the remainder of the process, while their quadtrees are updated continuously during rendering of the scene. In addition, our data structure represents a *sparse* set of pixels in need of anti-aliasing, and we do not require a strict front-to-back traversal of all triangles [16].

5.3 Sparse Anti-Aliasing

This subsection describes how our sparse visibility computations, culling, and shading are done (rightmost gray box at the bottom in Figure 2). The purpose of this stage is to efficiently resolve visibility among the triangles of a tile and ultimately calculate the color of each important pixel within. All triangles within a tile are processed in submission order. This stage defines sequence points where the blend mode, z-mode, or other pixel back-end state changes. All triangles contained within the same sequence points are treated as a continuous *triangle sequence*,

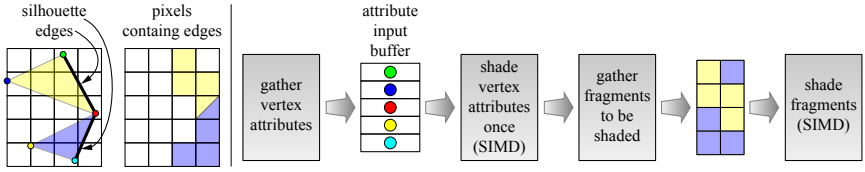


Figure 4: *Left: two triangles belonging to the same draw call request that some fragments be shaded since they have silhouette edges in several pixels. Note that one pixel needs to shade both the yellow and the blue triangle. Right: the shading pipeline. First, the vertex attributes are gathered into an input buffer, and then these are shaded in parallel using SIMD execution. Next, the fragments to be shaded are gathered, and finally, these fragments are shaded (which includes attribute interpolation). Note that in reality, more triangles are usually present. For example, one triangle would usually exist between the blue and yellow triangle–we excluded it to make the figure clearer.*

even if triangles belong to different draw calls.

Our current approach to high-quality anti-aliasing is simply adaptive multi-sampled anti-aliasing (MSAA), where several visibility samples are used per pixel and triangle, but only a single shading sample. We have found 16 visibility samples per pixel to be sufficient for most scenes, but this can be specified on a per-pixel level, and can even vary across the image. First, inside testing is performed against the edge equations of a triangle, and for each visible sample, a depth value is computed and a triangle pointer is stored.

Shading is delayed to a point where it is absolutely necessary, generally reducing the amount of shading performed. In the case of opaque geometry, the resolve defers shading within the tile until a triangle sequence with a transparent blend mode starts, or until all triangles within the tile have been processed. In the case of transparent geometry, fragments are shaded for a small group of triangles at a time, e.g., 4 or 8. They are then blended with the tile's samples in submission order. More details about the shading pipeline can be found in Section 5.3. However, first, we describe a number of different culling methods that we use before shading is done.

Culling

Depending on the rendering state of the triangle sequence, different optimizations can be applied. In the common case, where a triangle sequence represents opaque geometry with depth testing enabled, all optimizations are active. First, all triangles whose $z_{min}^{tri} > z_{max}^{tile}$ are removed from the triangle list of the tile. The remaining triangles are reordered to maximize occlusion culling within each pixel. In our implementation, the triangles are sorted based on z_{min}^{tri}.

To allow for occlusion culling at the pixel level, a z_{max}^{pixel}-value, representing the maximum depth of the samples in a pixel, is maintained. Once all samples in the pixel have been covered by opaque geometry, the value is used for occlusion culling. When a triangle in a triangle sequence is found with $z_{min}^{tri} > z_{max}^{pixel}$, the rest of the triangle sequence can be discarded since the triangles are sorted along z_{min}^{tri}. This scheme allows for efficient occlusion culling even when using highly tessellated models.

For opaque geometry, there are also a number of methods to avoid unnecessary shading, and hence can be seen as shading culling techniques. For example, if a single fragment is to be shaded for an important pixel, CPU shading is omitted since it would result in the same shading as already generated by the GPU. As a consequence, the pixel is discarded. Similarly, if there are two fragments that belong to two front-facing triangles *sharing* an edge, which obviously is not a silhouette edge, the corresponding important pixel is discarded. Finally, if there are multiple fragments, but none of the fragments belong to a triangle with a silhouette edge, then there cannot be any visible silhouette edge within the pixel. Hence, the important pixel is discarded. The last optimization assumes that there are no intersecting triangles in need of AA. Therefore, it cannot be applied if the important pixel is a manual addition.

Note that there is a small risk of increased aliasing when the optimizations from the previous paragraph are used. Since fragments are created from discrete samples, there is a possibility that a small triangle ends up between samples when our algorithm is used. The GPU pipeline, on the other hand, might sample this small triangle, and hence produce an aliased pixel, which is different from the resulting pixel from our high-quality rasterizer. The risk of this happening can be reduced to numerical differences by specifying one of our sample points to coincide with the sample location used by the GPU. However, this issue is ultimately a sampling problem that can be solved by increasing the sample rate of our CPU pipeline.

Shading

Self-contained fragment shaders, such as those using static textures and uniforms, are trivial to implement in the CPU pipeline. More advanced shaders, in particular those that use render targets (RTs), may require a more elaborate solution, depending on the underlying architecture. When memory is shared between the GPU and the CPU, RTs (created by the GPU pipeline) are directly accessible to the CPU pipeline. However, when the CPU and the GPU are separated by a relatively slow memory bus, issuing transfers of all RTs from the GPU to the CPU each frame, is prohibitively expensive. One solution is to evaluate RTs lazily when they are accessed by a CPU shader. However, even though the evaluation becomes sparse, it would become expensive with many RTs. It is also unclear how lazy evaluation could be implemented in a feed-forward rasterizer. For these reasons, we focus on shared memory architectures in this paper, and we note that with a shared memory architecture many more different flavors of an algorithm are possible and can

become efficient.

When possible, shading is performed for all deferred samples within the tile simultaneously. This enables SIMD execution over fragments from different important pixels and will thus have better efficiency than traditional quad rendering in our sparse setup. Initially, the samples within each pixel are compacted into per-pixel per-triangle fragments with a sample mask. These fragments are then sorted based on draw call, triangle, and pixel, in that order. All fragments belonging to the same draw call are processed simultaneously so that a single shader invocation can be performed for all draw call fragments. First, all vertex attributes for the draw call triangles are fetched into an input buffer. Then, all attributes are shaded using a single attribute shader invocation. Note that the attributes for each triangle are shaded exactly once within a tile. The resulting shaded attributes are then interpolated to the individual fragments. Finally, the interpolated attributes are used as input to a single fragment shader invocation that calculates the color of each fragment. The resulting colors are then distributed to the individual samples and they are marked as shaded. An example of our shading pipeline is shown in Figure 4. When all triangles have been processed in a tile, the final pixel color is computed as the mean of all the samples' color, which is the last step of our pipeline.

Next, we describe how derivatives are computed in our pipeline. A common way to compute attribute derivatives for, e.g., texture mip map selection, is to render quad fragments and estimate the derivatives as the differences between the samples. In our setup, this would be very inefficient since we are rendering a sparse set of pixels. Therefore, we employ analytical derivatives of the barycentric coordinates. These derivatives can simply replace the ordinary barycentric coordinates for attributes that are to be differentiated, rather than interpolated, before fragment shading. One issue that this fails to deal with is derivatives of indirect texture lookups. In this case, we resort to performing multiple lookups inside the fragment shader in order to perform the differentiation explicitly.

6 Implementation

This section describes our high performance CPU pipeline that implements all the pipeline stages described in Section 5. It is written specifically for the x86-64 architecture targeted at multicore CPUs supporting SSE 4.1 instructions at a minimum. Most of our implementation is independent of the architecture's SIMD width and simply process more data in Struct of Arrays (SoA) fashion. This means that the same algorithm can use either 4-wide SSE or 8-wide AVX, as well as future SIMD widths, without much adjustment. The inputs to our pipeline, such as vertex buffers, contain regular Array of Structs (AoS) data. Therefore, we make frequent use of register transpose to convert loaded AoS data to SoA form.

Our pipeline makes use of a rasterizer based on two-dimensional edge equations from projected and grid-snapped vertices, with similarities to Larrabee rasterization [1]. While this makes it possible to perform rasterization using fixed point

math, we still resort to AVX floating point computations for efficiency. This is fine as long as all floating point values are within ranges that guarantee an exact result. For example, a 32-bit IEEE 754 floating point value can represent a 24-bit fixed point value exactly (not counting the sign bit). In the future, we will investigate using AVX2 since it extends integer arithmetic to 256 bits.

In order to distribute work among multiple CPU cores, the pipeline splits the different stages of Section 5 into various *tasks* that are consumed by a pool of *worker threads*. The *main thread* can queue multiple independent tasks for processing. Each task has a number of *work items* that represent the basic unit of work distributed to the worker threads. The work items in each task have a global order; the first work item of the second task is numbered immediately after the last work item of the first task. This allows all worker threads to acquire their work in a lock-free fashion by using an atomic increment of a shared counter. Each task can spawn new tasks when finished and can thus implement dependency chains. The main thread have the ability to wait for all tasks to finish in order to synchronize different stages.

The vertex position shading stage is trivially implemented by processing multiple vertices simultaneously in SIMD fashion. Each task consists of a single draw call, and each work item consists of a subset of vertices to shade. Once a vertex shading task completes, it immediately queues a task for performing triangle setup for each triangle. Triangle setup tests for trivial rejection against the view frustum, performs clipping, snaps its vertices to a two-dimensional grid, determines its facing, computes a bounding box, and sets up edge equations. The facing information is used to speed up the silhouette detector, which follows in the dependency chain.

The silhouette detection task checks front facing triangles for silhouettes by testing the facing of adjacent triangles. The resulting silhouette edges are clipped against the view frustum and conservatively rasterized to the importance map. In order to avoid the synchronization necessary to set individual bits in the importance map, each entry is instead byte sized. Note that vertex shading, triangle setup, and silhouette detection are performed independently for all draw calls without any synchronization. Before starting the next stage, the main thread waits for all outstanding tasks to finish. Once all draw calls have been shaded, and all silhouettes have been rasterized to the importance map, the HIM is built. First, a task that builds tiled subsets of the HIM is executed in parallel. Then a single thread builds the highest levels of the tree over the generated tiles. The HIM is used when binning triangles in the next stage.

The sparse rasterization stage creates one task for each draw call, where a subset of a draw call's triangles are processed in each work item. Each work item reads 4 or 8 front facing triangles at a time, depending on the SIMD width of the CPU, and stores them in vector registers in SoA form. Active triangles are indicated by a *lane mask*, which is updated as triangles get rejected. A bounding box is then computed around the active triangles, in order to allow for an immediate jump to the lowest possible level in the HIM. This is done efficiently as `int lvl=32-clz(max(bbw-1,bbh-1))`, where `bbw` and `bbh` are the bound-

ing box width and height, respectively, and `clz()` is an instruction that counts the number of leading zeroes. Note that level 0 is the highest resolution level in the HIM. All active triangles are then simultaneously traversed against the HIM, while maintaining active triangles in the lane mask. When the tile level is reached, z_{min}^{tri} and z_{max}^{tri} are computed and all triangles, which are currently active in the lane mask, are written to the tile's triangle list, after testing for occlusion. In order to avoid synchronization, each thread has its own per-tile triangle list.

Once all triangles have been binned to tiles, the sparse anti-aliasing stage creates a single task with work items corresponding to a tile each. Each work item starts by reserving and clearing samples for all important pixels within the tile. Then, the draw call triangles overlapping the tile are read in submission order, by pick sorting from the per-thread triangle lists created in the previous stage. Once a whole triangle sequence have been read, and possibly reordered, the triangles are sample tested. First, the edge equations for a SIMD width of triangles are read. Then, a lookup table, containing the result of evaluating the edge equations at the local sample positions, is set up in order to accelerate the sample tests. For each important pixel, all loaded triangles are conservatively tested against the pixel extents in parallel. Then, each triangle, possibly overlapping the pixel, is sample tested one at a time. Inside- and depth testing is performed at the sample locations using the previously computed lookup table, offset by the edge equations evaluated at the pixel. This is done in parallel over samples. Shading is computed exactly as described in Section 5.3. Note that since the CPU cores work in parallel with the graphics processor, our AA algorithm does not increase the frame latency, which is a highly desired feature.

Our code base is rather extensive, and in order for others to be able to reproduce the results and make use of our algorithm, the source code of our CPU rasterizer is released under the MIT license.

7 Results

For all our results, we have used an Intel Ivy Bridge Core i7 (3770K) at 3.5 GHz with four cores (eight threads) and with an integrated graphics processor (HD Graphics 4000) containing 16 EUs (execution units) for unified shading. The EUs are capable of a theoretical 166.4/294.4 GFLOPS without/with Turbo, while the CPU cores have a theoretical peak of 224/249.6 GFLOPS. The thermal design power (TDP) for this chip is 77 W, including both the CPU *and* the GPU. Our machine uses Windows 7 as operating system, and we have implemented our algorithm using DirectX 11 with the *DirectResourceAccess*[1] extension for shared memory from Intel.

Four different scenes have been used for our experiments. These are *Chess*, *Sponza*, *Buddha*, and *Hairball*. Chess and Hairball both have Phong shading and shadow

[1] Also called *InstantAccess*.

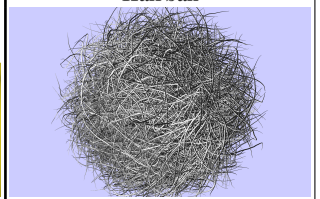
Scene	**Chess**			**Sponza**			**Buddha**			**Hairball**		
												
# triangles	64k			262k			1.1M			2.85M		
resolution	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
% imp. pixels	1.6%	1.0%	0.7%	4.4%	2.7%	1.9%	1.5%	0.8%	0.5%	31%	21%	15%
WARP 4 spp	25 ms	55 ms	107 ms	62 ms	114 ms	208 ms	154 ms	225 ms	349 ms	319 ms	418 ms	530 ms
WARP 8 spp	37 ms	90 ms	169 ms	81 ms	163 ms	310 ms	188 ms	277 ms	433 ms	399 ms	564 ms	761 ms
A4 16 spp 100%	26 ms	56 ms	101 ms	50 ms	104 ms	186 ms	96 ms	156 ms	247 ms	126 ms	182 ms	254 ms
A4 8 spp	5 ms	9 ms	15 ms	15 ms	22 ms	31 ms	34 ms	40 ms	52 ms	110 ms	149 ms	193 ms
A4 16 spp	5 ms	9 ms	15 ms	16 ms	23 ms	34 ms	37 ms	40 ms	52 ms	119 ms	160 ms	207 ms
GPU 1 spp	5 ms	7 ms	12 ms	8 ms	12 ms	18 ms	25 ms	33 ms	44 ms	40 ms	52 ms	66 ms
GPU 4 spp	7 ms	13 ms	20 ms	14 ms	23 ms	37 ms	30 ms	41 ms	55 ms	64 ms	111 ms	169 ms
GPU 8 spp	10 ms	20 ms	35 ms	20 ms	37 ms	59 ms	34 ms	48 ms	63 ms	112 ms	200 ms	300 ms

Table 1: *Statistics for rendering of our four test scenes, where* $R_1 = 1280 \times 800$*,* $R_2 = 2048 \times 1280$*, and* $R_3 = 2880 \times 1800$ *are the resolutions that we have used. The third row shows the percentage of important pixels after culling in the rendered images, and below that, results are shown in time (milliseconds) per frame. Note that asynchronous adaptive anti-aliasing (*A4*) method uses 8 and 16 samples per pixel (spp), while both WARP and the GPU only use 4 and 8 spp MSAA. In addition, the row denoted* A4 *16 spp 100% also uses our algorithm with the difference that* ***every*** *pixel is rendered using the CPU pipeline.*

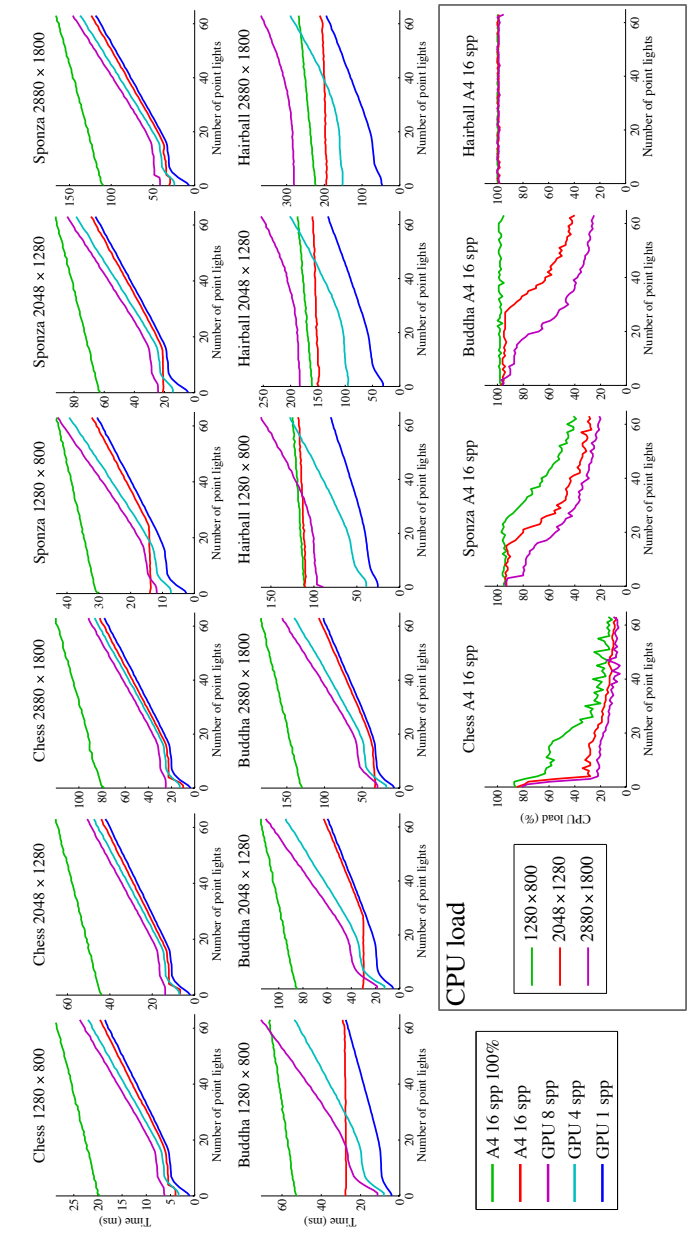


Figure 5: *All scenes rendered at different resolutions using a variable number of diffuse point light sources. Each plot represents one scene at one resolution rendered using both A4 and the GPU at different settings. The number of point lights increase along the x-axis, and the execution time is indicated by the y-axis. In the framed region at the bottom, CPU load over the number of light sources is visualized for all configurations of A4 16 spp. CPU usage was measured by polling performance counters provided by the operating system.*

mapping with one light source. Sponza also has Phong shading and one shadow map, but also texturing and alpha-textured transparent flowers. Silhouette detection was disabled for the flowers, because their textures are transparent towards the edges. This is optional, however. The Buddha scene has four Phong shaded light sources, each with an individual shadow map. Percentage-closer filtering (PCF), for shadow mapping, was implemented in the shader by interpolating four nearest neighbor samples.[2] We compare our asynchronous adaptive anti-aliasing algorithm, which is denoted A4 and is described in Sections 3-6, against both the integrated GPU and against WARP [15], which is an optimized software rasterizer used as a fallback in Windows 7. For the GPU and WARP, we use MSAA with 4 and 8 spp, where 8 spp is the highest setting for these two renderers. For A4, results are shown for both 8 and 16 spp. We have used sampling patterns which are digital nets for high quality. For example, for 16 spp, we use a $(0,4,2)$-net in base 2. The results have been generated for 1280×800, 2048×1280, and 2880×1800 resolutions, where the latter is the native resolution of a Macbook Pro 15" (2012). We use WARP as a reference for software rendering performance. In practice, A4 has the advantage of letting the GPU render all shadow maps. In order to make the comparison more informative, we do not count the time WARP needs to render the shadow maps. Note that this gives WARP an advantage over A4 in that it does not need to wait for the GPU to finish shadow map rendering.

Our main results are shown in Table 1. For A4, we have used a tile size of 8×8, since that size gave best results for all scenes and resolutions. One row in the table says "A4 16 spp 100%", which means that we deliberately set all pixels to be important in our algorithm. As a consequence, all pixels are rendered using the CPU pipeline for that setting. It is interesting to see that with 16 spp, A4 100% is about on par with WARP 4 spp for Chess, but for the rest of the scenes, A4 100% is significantly faster, despite the fact that WARP does no shadow map rendering and A4 has overhead for silhouette detection, HIM generation & construction, and is optimized for sparse rendering & shading. In addition, A4 100% has $4\times$ as many spp. For the hairball, A4 100% is more than twice as fast. Note also that our adaptive A4 algorithm with 16 spp is between 3.9–7.1$\times$ faster than WARP at 4 spp, for the first three scenes, and for the hairball, this is reduced to 2.6-2.7$\times$ faster. Hence, our method generates higher quality images and is faster than WARP.

The other interesting results are just below the A4 100% row, where the numbers for our A4 algorithm are shown together with MSAA numbers for the GPU. Note, for example, that A4 with 8 spp always is as fast or faster than the GPU with 8 spp, for all resolutions and all scenes, and the gap is bigger, the higher resolution. In fact, for the highest resolution, A4 is 1.2–2.3$\times$ faster than the GPU. Comparing A4 with 16 spp vs. GPU with 8 spp, it can be seen that A4 is faster or about the same. Also, our algorithm scales better than the other algorithms with increasing resolution. This is expected since the percentage of the pixels with silhouette edges decreases with higher resolution. We also note that A4 with 8 spp is exactly as fast

[2]Hardware PCF could not be combined with R32 float render targets, which we use for shadow maps on our current platform.

	Chess	Sponza	Buddha	Hairball
Vertex Position Shading	1.4%	1.7%	4.0%	2.1%
Triangle Setup	6.6%	5.9%	21.0%	13.5%
Silhouette Detection	3.1%	3.5%	10.0%	8.3%
HIM Build	3.1%	0.6%	0.7%	0.1%
Sparse Rasterization	14.4%	16.3%	19.2%	17.3%
Sparse AA - Visibility	40.9%	37.1%	28.4%	36.3%
Sparse AA - Shading	23.0%	30.8%	15.9%	21.3%
Copy to frame buffer	7.6%	4.1%	0.9%	1.1%

Table 2: *The time spent in the different stages of our algorithm.*

or just a little bit faster than A4 with 16 spp, which is a consequence of that we have focused on optimizing the 16 spp variant more.

The time spent in the different stages of our pipeline is summarized in Table 2 for our test scenes at 2048×1280. As can be seen, most of the time is spent in the later stages of the pipeline, much like a real GPU pipeline. The Buddha scene is a bit different and spends more time in the earlier stages. This is expected since Buddha has very few silhouettes compared to the number of triangles in the scene, which means that most work is culled before the later stages. It is also interesting to look at the CPU load using our algorithm, e.g., A4 with 16 spp for the scenes in Table 1. For Chess, it varies from 52% (highest resolution) to 72% (lowest resolution). For Sponza, the corresponding numbers are 84%-91%, and for Buddha: 72%-89%. The hairball is more extreme, due to the many silhouettes, and the CPU load is about 99% all the time. The CPU load is related to how much A4 needs to be idle and wait for the GPU to render shadow maps and the frame buffer at 1 spp. The fact that the CPU load generally decreases with higher resolution is evidence that the GPU becomes the bottleneck at higher resolution.

To determine how A4 scales with increasing shader complexity, a shader with varying number of diffuse point lights and no textures/shadow maps, was applied to all scenes. The results are shown in Figure 5, where it is clear that our algorithm just becomes a small offset to GPU 1 spp at some point. One interpretation for this is that more expensive shaders are beneficial to our algorithm. This and better resolution scaling are two really important features of A4. The results also indicate what factors contribute to increased rendering time when using MSAA, and hence how A4 can improve performance. The difference in time at zero point lights can be interpreted as the cost of additional visibility computations, while the slope of the curve indicates the amount of extra shading. The amount of redundant shading on the GPU due to quad shading is hard to estimate, but we note that A4 100% scales best with increased shading complexity. This behavior can be attributed to, e.g., reduced quad shading overhead and better occlusion culling.

Next, we will discuss some disadvantages with our method. First, our current implementation does not handle tessellation. This would require our vertex position shading stage to also tessellate, compute the positions of the newly created tri-

angles, and then detect silhouette edges after that. This would require accurate knowledge about how the hardware tessellator works, and therefore, it is left as future work. Second, for best results, surfaces should not intersect. This can, however, be avoided by the artists at the cost of time. Third, our algorithm is designed around the fact that many scenes have rather few important pixels, which need high-quality AA. A difficult scene is therefore the hairball (right in Table 1), as seen in Figure 6. However, the results here were surprising. A4 (8/16 spp) rendered the scene as fast or faster than the GPU at 8 spp. In fact, for the two higher resolutions, A4 100% was also faster than the GPU at 8 spp.

Finally, we have also measured image quality, in terms of peak signal to noise ratio (PSNR), at 2048×1280 against a ground truth image rendered with 256 samples per pixel with MSAA. The results are shown below in dB (decibels), i.e., higher numbers are better.

	Chess	**Sponza**	**Buddha**	**Hairball**
GPU 4 spp	48.9 dB	48.5 dB	50.1 dB	33.1 dB
GPU 8 spp	52.8 dB	51.9 dB	53.6 dB	37.1 dB
A4 16 spp	54.9 dB	47.1/52.0 dB	54.9 dB	40.1 dB

As expected, the PSNR is substantially higher for A4 using 16 spp compared to the GPU using 4 & 8 spp. The exception is Sponza, where the PSNR drops to 47.1 dB, which is mostly due to our current handling of alpha-textured geometry. If transparent geometry also generates silhouettes, PSNR increases to 52.0 dB, which is a bit better than the GPU at 8 spp. Even though the geometry gets transparent toward the edges, there are still some hard edges that could be resolved with more careful texture mapping. Note also that the rendering time increases from 23 to 34 ms when transparent geometry generates silhouettes, which is still faster than the GPU at 8 spp. This rather substantial increase comes from the fact that shader culling will be inefficient for all tiles with overlapping transparent geometry in our current implementation. There is no way to cull before the alpha geometry has been rendered, and at that point, the tile has already shaded everything for blend operations to work. We note that this could be resolved by having conservative z_{min} for all silhouettes within an important pixel, and perform shader culling if $z_{min}^{silhouette} > z_{max}^{pixel}$, but leave this to future work.

As mentioned earlier, it is possible to apply our algorithm to *only* as many important pixels as there is time before the target frame time has been reached. Recently, Guenter et al. [19] used eye tracking to render in high resolution only in a small gaze region of the viewer's eyes. With our algorithm, we could use a similar system, but instead increase the sampling rate only in the gaze regions, for faster rendering. This is left for future work, but it gives some confidence that our platform provides new and interesting alternatives. The energy efficiency of A4 is also left for future work, i.e., since our approach uses both the graphics processor and the CPU cores in the chip, it may be that more energy is used compared to using only $16\times$ MSAA on the graphics processor. However, since our method used less time to solve the problem, it is possible (but not certain) that our method also uses less energy.

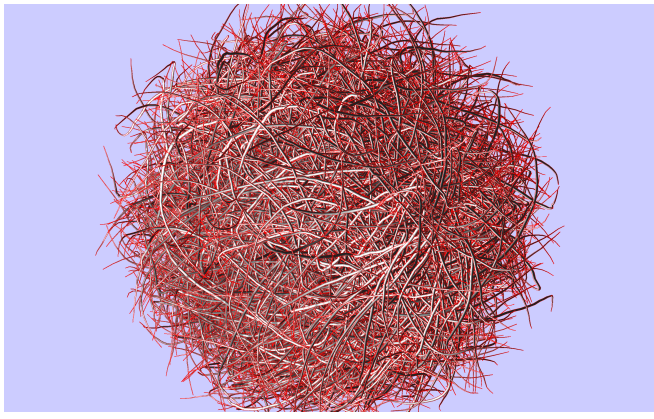


Figure 6: *Hair ball rendered with* A4 *with* 16 *spp in 207 ms at* 2880×1800*. This is a difficult case for our algorithm, since there are many silhouette edges. All important pixels (after all sorts of culling) are marked as red. The* GPU *used 300 ms with 8 spp MSAA.*

8 Conclusions and Future Work

We have presented a real-time hybrid rasterization pipeline, which uses both the CPU and its integrated graphics processor, for adaptive edge anti-aliasing. The key hardware component that glues together the communication and sharing of resources between the GPU and the CPU is a shared memory architecture. Our results are very promising, and in particular, we believe that our results shows that a shared memory architecture can be of great benefit to researchers and developers. In addition, the resolutions of commodity devices have been increasing rapidly over the last few years, and our algorithm scales better with higher resolutions. This makes it an interesting alternative for integrated CPU and GPU architectures, now and for even higher display resolutions (e.g., 3k and 4k).

Most GPU research has focused on discrete graphics cards, but we believe that shared memory architectures open up for a wide range of novel rendering algorithms. Our system can be seen as a starting point for a more general platform for hybrid rendering, and in the future, we would like to investigate other uses, e.g., ray tracing. We also want to explore whether the sampling rate can be increased adaptively when a small triangle is missed inside a pixel. Furthermore, we also want to incorporate analytical [11, 7] and semi-analytical methods [8] into our renderer. It may, e.g., be possible to develop an analytical method that handles 2–4 triangles per pixel, and then revert to MSAA for more triangles. For future work, we will also attempt to augment Crow's method to motion blur and depth of field.

Acknowledgements

Thanks to Aaron Lefohn, Charles Lingle, Axel Mamode, Petrik Clarberg, Richard Huddy, and Tom Piazza at Intel. The Buddha comes from the Stanford Computer Graphics Laboratory, Hairball is courtesy Samuli Laine, and Sponza is courtesy of Marko Dabrovic/Frank Meinl. Tomas is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

Bibliography

[1] ABRASH, M. Rasterization on Larrabee. *Dr. Dobb's Journal* (May 2009). http://www.drdobbs.com/parallel/rasterization-on-larrabee/217200602.

[2] AILA, T., MIETTINEN, V., AND NORDLUND, P. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics 22*, 3 (2003), 792–800.

[3] AKENINE-MÖLLER, T., AND AILA, T. Conservative and Tiled Rasterization Using a Modified Triangle Set-Up. *Journal of Graphics Tools 10*, 3 (2005), 1–8.

[4] AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. *Real-Time Rendering*, 3rd ed. AK Peters Ltd., 2008.

[5] AMANATIDES, J., AND WOO, A. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics* (1987), pp. 3–10.

[6] ANDERSSON, J. Five Major Challenges in Real-Time Rendering. In *Beyond Programmable Shading course* (2012), SIGGRAPH.

[7] AUZINGER, T., WIMMER, M., AND JESCHKE, S. Analytic Visibility on the GPU. *Computer Graphics Forum (Proceeding of EUROGRAPHICS 2013) 32*, 2 (2013), 409–418.

[8] BARRINGER, R., GRIBEL, C. J., AND AKENINE-MÖLLER, T. High-Quality Curve Rendering using Line Sampled Visibility. *ACM Transactions Graphics 31*, 6 (2012), 162:1–162:10.

[9] BERGERON, P. A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics and Applications 6*, 9 (1986), 17–28.

[10] CARPENTER, L. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (1984), pp. 103–108.

[11] CATMULL, E. A Hidden-Surface Algorithm with Anti-Aliasing. In *Computer Graphics (Proceedings of SIGGRAPH 78)* (1978), pp. 6–11.

[12] CROW, F. Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of SIGGRAPH 77)* (July 1977), pp. 242–248.

[13] CROW, F. C. The Aliasing Problem in Computer-Generated Shaded Images. *Communications of the ACM 20*, 11 (1977), 799–805.

[14] FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. Reducing Shading on GPUs using Quad-Fragment Merging. *ACM Transactions on Graphics 29* (2010), 67:1–67:8.

[15] GLAISTER, A. Windows Advanced Rasterization Platform (WARP) Guide. MSDN, November 2008.

[16] GREENE, N. Hierarchical Polygon Tiling with Coverage Masks. In *Proceedings of SIGGRAPH* (1996), pp. 65–74.

[17] GREENE, N., AND KASS, M. Error-Bounded Antialiased Rendering of Complex Environments. In *Proceedings of SIGGRAPH* (1994), pp. 59–66.

[18] GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH* (August 1993), pp. 231–238.

[19] GUENTER, B., FINCH, M., AND SNYDER, J. Foveated 3D Graphics. *ACM Transactions on Graphics 31*, 6 (2012), 164:1–164:10.

[20] JIMENEZ, J., GUTIERREZ, D., YANG, J., RESHETOV, A., DEMOREUILLE, P., BERGHOFF, T., PERTHUIS, C., YU, H., MCGUIRE, M., LOTTES, T., MALAN, H., PERSSON, E., ANDREEV, D., AND SOUSA, T. Filtering Approaches for Real-Rime Anti-Aliasing. In *ACM SIGGRAPH 2011 Courses* (2011).

[21] MOREIN, S. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings* (August 2000).

[22] PIAZZA, T. Processor Graphics. In *High Performance Graphics – Hot3D Talks* (June 2012).

[23] PINEDA, J. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)* (August 1988), pp. 17–20.

[24] RESHETOV, A. Morphological Antialiasing. In *High Performance Graphics* (2009), pp. 109–116.

[25] SANDER, P. V., HOPPE, H., SNYDER, J., AND GORTLER, S. J. Discontinuity Edge Overdraw. In *Symposium on Interactive 3D Graphics* (2001), pp. 167–174.

[26] WARNOCK, J. W. A Hidden Surface Algorithm for Computer Generated Halftone Pictures. Tech. rep., Utah University, 1969.

[27] YOUNG, P. Coverage-Sampled Anti-Aliasing. Tech. rep., NVIDIA, 2007.

Paper V

Dynamic Ray Stream Traversal

Rasmus Barringer Tomas Akenine-Möller

Lund University

ABSTRACT

While each new generation of processors gets larger caches and more compute power, external memory bandwidth capabilities increase at a much lower pace. Additionally, processors are equipped with wide vector units that require low instruction level divergence to be efficiently utilized. In order to exploit these trends for ray tracing, we present an alternative to traditional depth-first ray traversal that takes advantage of the available cache hierarchy, and provides high SIMD efficiency, while keeping memory bus traffic low. Our main contribution is an efficient algorithm for traversing large packets of rays against a bounding volume hierarchy in a way that groups coherent rays during traversal. In contrast to previous large packet traversal methods, our algorithm allows for individual traversal order for each ray, which is essential for efficient ray tracing. Ray tracing algorithms is a mature research field in computer graphics, and despite this, our new technique increases traversal performance by 36-53%, and is applicable to most ray tracers.

ACM Transactions on Graphics, 33(4):151:1–151:9, 2014.

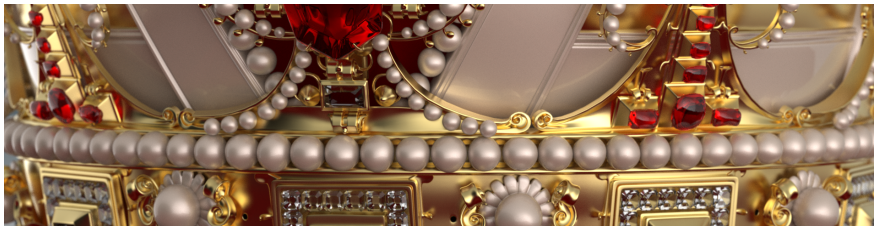


Figure 1: *A close-up of the crown scene rendered using Embree 2.0 modified with our novel traversal algorithm. This scene renders, including shading, frame buffer updates, and all other overhead, about 22% faster with our algorithm on a Haswell-based ultrabook. The performance improvement comes from our traversal, which is 36% faster than the fastest traversal algorithm in Embree for this scene. For other scenes, our traversal algorithm can be over 50% faster.*

1 Introduction

Ray tracing [27] is a flexible tool that forms the basis for state of the art photo-realistic rendering algorithms. Recently, it has also seen increasing use in real-time applications. One of the core operations in a ray tracer is determining the closest visible surface along a ray. The performance of this operation is critical and is usually accelerated by building a spatial data structure over the scene. One of the most popular spatial data structures is the bounding volume hierarchy (BVH). The time required to trace a set of rays against a BVH is dependent on the size of the scene, the distribution of the rays, the quality of the BVH, and the performance of the ray traversal algorithm. Our goal is to create a high-performance ray traversal algorithm that is less sensitive to the size of the scene and the distribution of the rays, compared to other packet tracing approaches.

Modern CPUs employ a variety of techniques to improve performance by rearranging programs to exploit inherent parallelism [12]. Superscalar CPUs can execute multiple instructions simultaneously on different functional units and pipelining is used to issue a new instruction each clock cycle. This makes it possible to realize high throughput with many independent instructions, even though the latency of different instructions may differ. When the CPU is unable to issue a new instruction, resources are wasted. Therefore, instruction scheduling is performed both by the compiler and the CPU. However, efficient scheduling is possible if and only if the algorithm contains enough independent instructions at any given time. Our goal is to construct a ray traversal algorithm that aligns well with current hardware, by providing enough parallel work within a single instruction stream.

Data level parallelism attempts to make the most out of each instruction by performing the same computation on several data items using single instruction multiple data (SIMD) execution. The extra performance scales linearly with SIMD width for suitable algorithms and is very power efficient. Current trends indicate

that SIMD width will continue to increase in the future. For example, contemporary CPUs have a SIMD width of 256 bits, the many-core Xeon Phi architecture features 512-bit SIMD, and most GPUs have a width of 1024 or 2048 bits. Algorithms need to have low instruction level divergence in order to efficiently exploit increasing SIMD width. Divergence leads to underutilization since SIMD lanes will need to be masked out. This is not a trait generally attributed to depth-first traversal. Even when packets of rays are traced in a SIMD fashion, rays usually diverge quickly when incoherent, such as for diffuse interreflections, for example.

As compute increases (wider SIMD and more cores), one of the key challenges involves the external memory bandwidth and cache design. To reduce latency and bandwidth, CPUs are equipped with a large cache hierarchy in combination with sophisticated hardware prefetchers that detect memory access patterns, and prefetch memory into a cache before it is needed. Our algorithm is designed to have a predictable memory access pattern with high data coherence, which substantially reduces the amount of memory bandwidth usage in our tests.

Over the last 30 years or so, the topic of ray tracing has been researched thouroughly, and can in many ways be considered a mature research field. As a consequence, it is increasingly difficult to develop algorithms that improve performance, and in particular so for the core methods, such as traversal, intersection, and shading. Despite this, our results show that total ray tracing performance can be improved by 22–37%, while traversal alone is increased by 36–53%, which is rather remarkable. In addition, we expect that our techniques can be applied to a wide range of existing ray tracers, since the BVH is the most popular spatial data structure.

2 Previous Work

As mentioned above, a substantial amount of research has been devoted to finding new and improved ray traversal algorithms in order to make the entire ray tracing process faster. Here, we can only review a small number of these papers, and we chose the ones that are most relevant for our work.

Usually, a stack is maintained that contains the next node to be processed during ray traversal. The technique has been combined with various forms of ray sorting and tracing of whole packets [26] of rays to improve performance. Sorting generally incurs some overhead and is usually a heuristic that hopes to improve data locality and minimize divergence. Packets of rays are often quite small and inevitably leads to divergence in all but the most coherent workloads (such as primary visibility). Stack-less ray traversal [13, 16, 11, 2] is a more recent endeavor that was, at least initially, motivated by the high overhead of maintaining a traversal stack on previous generations of GPUs. More recently, uses have been postulated to include cheap ray suspension and transfer on distributed systems or custom hardware [11], though no real demonstration of such system exists, to the best of our knowledge. Áfra and Szirmay-Kalos [1] present optimized stack-less traversal algorithms for CPUs, MIC, and NVIDIA GPUs using multi-BVHs [24, 7, 5],

where a BVH node usually have four or more children in order to improve SIMD efficiency in ray vs. many bounding volumes and intersection tests.

While both stack and stack-less traversal generally performs a depth-first traversal with a small number of rays, Hanrahan [10] introduced breadth-first ray tracing to increase the number of cache hits for geometry in a beam tracer. Stream filtering [25, 9, 22, 21] builds upon the ideas of breadth-first ray tracing to test a large number of rays against the same BVH node or triangle, which may allow high utilization of very wide SIMD. The downside of their approach is mainly that all rays need to traverse the BVH in the same order. Mora [19] traverses large packets of rays against a set of triangles, but perform traversal in such a way that a BVH is not needed.

When traversing a binary BVH at a certain node, and it is determined that a ray intersects both children, the algorithm must determine which node to traverse next, and which to postpone (usually by pushing it to a stack). A *very important* optimization in ray tracing is to let each ray start traversing the child node that is most likely to occlude the ray, thus potentially making traversal into the other child unnecessary. The order is often based on some heuristic, e.g., start with the node closest to the ray origin. One major disadvantage in existing stream filtering approaches is that all rays must start with the same node, meaning that, for highly divergent rays, about half of the rays will start with a suboptimal node. Our approach combines the strengths of stream filtering, in terms of SIMD efficiency and memory locality, with the flexibility of a depth-first traversal so that any ordering heuristic is possible for each individual ray.

Boulos et al. [4] use ray packets of arbitrary size in order to exploit coherence otherwise hidden in separate smaller packets. They combine BVH packet traversal (interval arithmetic), SIMD packet tracing, and breadth-first ray tracing. However, they do not attempt to allow for dynamic descent direction for each individual ray during traversal, i.e., all rays in a packet must take the same path in the tree. Tsako et al. [22] combine multi-BVH traversal with stream tracing of large packets in order to reduce the memory bandwidth associated with single-ray tracing against an n-way BVH, while eliminating the cost of filtering rays at each traversal step. They maintain one stack of rays per SIMD lane that are referenced by a single task stack. Again, the traversal order is the same for all rays.

Garanzha and Loop [8] target GPU architectures and improve performance by efficiently sorting rays for coherence and then traversing coherent packets of rays using frustums in a breadth-first manner. They use an Octo-BVH and traverse using an ordering heuristic based solely on the average ray direction within a frustum.

3 Overview

As previously mentioned, a stack, which contains nodes that are still to be tested against the ray, is usually maintained during single-ray traversal. If multiple rays are traversed simultaneously, rays will typically have some of the same entries in

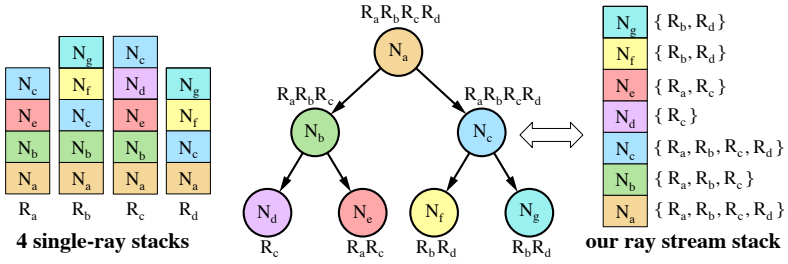


Figure 2: *Example traversal state showing the relationship between multiple single-ray stacks and our combined ray stream stack. $N_{a..f}$ represents nodes of the BVH and $R_{a..d}$ represents rays traversed against it. Left: four single-ray stacks – note that since a stack is a data structure that changes over time, we have chosen to just visualize which nodes are visited by each ray. Middle: here the BVH is shown, and next to each node, all the rays that visit that node are listed. Right: our ray stream stack is visualized here, and it is straightforward to confirm that those two stack representations include exactly the same information. One part of our new traversal algorithm is that we perform ray vs. node tests in the order provided by the ray stream stack to increase SIMD efficiency.*

their node stacks. This observation forms the basis for our algorithm. The key idea is that if multiple rays are to be tested against the same node, this can be performed in a very efficient manner. First, memory fetches for that node can be amortized over multiple rays. Second, since a single node is tested against multiple rays that are completely independent, we gain instruction level parallelism and have additional opportunities for vectorization. This means that 1) SIMD execution can be used, and 2) dependency chains that may stall the pipeline can be avoided. In fact, if enough rays are tested against the same node, vectorization can scale to any SIMD width.

We therefore form a *ray stream*, which is essentially just a collection of rays, e.g., the initial eye rays from a 16×16 pixel tile, or some light paths in a global illumination renderer. When traversing a ray stream against a BVH, our goal is to group rays that take the same path in the tree, without restricting the path of each individual ray. As such, we allow for a flexible traversal order but utilize coherence when present.

The subsequent section describes the details of the algorithm and then follows implementation details in Section 5. Results are presented in Section 6 and finally, we offer some conclusions and ideas for future work.

4 Traversal Algorithm

In order to extract coherence from individual rays in a ray stream, we remove the per-ray node stacks typically used in single-ray traversal. We instead define a single *ray stream stack* that is shared by all rays in a ray stream. In order to keep track of which rays should be tested against a particular node in the ray stream stack, each item in the stack has a list of pointers to associated rays. The relationship between multiple single-ray stacks and our ray stream stack is shown in Figure 2.

Ordinary stack-based tree traversal progresses in steps by testing a single ray, or a small packet of rays, against a node at a time. Our algorithm works in exactly the same way, only that, in our case, a variable sized set of rays is tested against the node at each traversal step. Testing the set of rays should ideally be performed in a loop over the set of rays with few dependencies between loop iterations, so that the loop can be unrolled to provide for instruction level parallelism and opportunities for vectorization. At the same time, we desire to group rays that continue along the same path so coherence can be continued to be exploited in the next traversal step.

Assume that an internal node contains pointers to its children, as well as the bounding volumes of the children. A high level description of our algorithm can be found below:

1. Build a ray stream of, e.g., 4096 eye rays.
2. Set *active rays* to be all rays in the ray stream. This is essentially a list of ray pointers or indices, referencing the ray stream.
3. Fetch the BVH root node, and make it *current node*.
4. Using SIMD, test all active rays against the current node:
   (a) If the node is internal, then test all rays against the BVs of the children (two in the case of a binary BVH).
   (b) If the node is a leaf, then test all rays against the triangles in the leaf.
5. According to the result from step 4a, push stack items to the ray stream stack, each with a BVH node and a corresponding ray pointer list of associated rays.
6. If the stack is empty, we are done. Otherwise continue with the next step.
7. Pop a stack item and make the item's BVH node the current node. Also make the associated list of rays the active rays.
8. Go to 4.

This algorithmic description is fairly complete, but omits an important optimization that switches to single-ray traversal as the active rays becomes few. How this optimization is applied is discussed in more detail in Section 5. The most interesting step in the algorithm is arguably step 5 that deals with appending rays to the ray stream stack. In fact, how to efficiently manage the ray stream stack and

its associated ray pointer lists during traversal is the topic of the remaining part of this section.

When testing a set of rays against an internal node (Step 4a), the outcome of the tests determines what stack items are added to the ray stream stack in Step 5. The maximum number of stack items that can be added, and to which child they each refer to, depends on how many children each interior node of the BVH has and the number of possible order configurations that are allowed for continued paths. For example, for BVH2 (Section 4.1), we need to push a maximum of $n = 3$ stack items to the ray stream stack. The specifics for BVH structures with 2 and 4 children are discussed in Sections 4.1 and 4.2.

In order to support a streaming approach where each ray is tested against the internal node once, we need an efficient way to append rays to the stack. The upper limit on the number of stack items that can be added, allows us to reserve space for all of them prior to testing any rays, i.e., before Step 4. After testing a specific ray (Step 4a) and determining how traversal should progress for that ray, a pointer to the ray is appended to all stack items that make up that particular path continuation. Once all rays have been tested, the result is a set of stack items that reference zero or more rays. Only those items that actually reference any rays are pushed to the ray stream stack in Step 5.

The limited number of stack items added at a traversal step means that memory for the ray pointer lists can be managed in a very efficient manner. If the maximum number of items is n, it is enough to allocate n large lists for ray pointers at program start. At each traversal step, ray pointers can simply be appended to the end of these lists by incrementing a counter. Because the ray stream stack always processes items in stack order, memory can be reclaimed from the n lists by subtracting the counter of each list after processing a stack item is complete. Note that we are usually reading ray pointers from one of the lists that we are also appending to. Given that we can never append more ray pointers than what we have read, a careful implementation can read and append new ray pointers in-place without any extra copying.

4.1 BVH2

In this subsection, we describe how our traversal algorithm works for a BVH2. Each internal node in a BVH2 structure has two children, namely one left and one right. The following outcomes are thus possible during the intersection test:

1. The ray misses both left and right children $\rightarrow$ do nothing.
2. Traverse into left child.
3. Traverse into right child.
4. Traverse into left child, then right child.

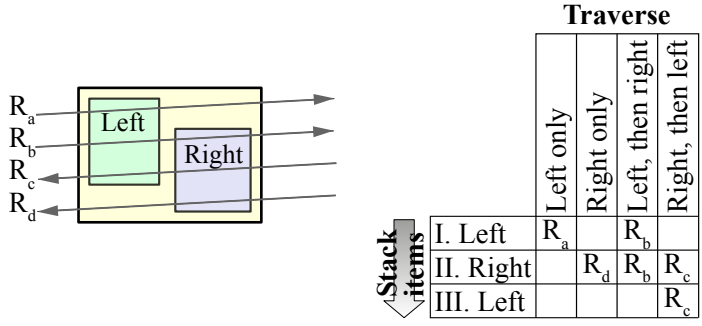


Figure 3: *To the left, a parent box is shown with a left and right child, and four rays,* R_a*,* R_b*,* R_c*, and* R_d*, which illustrate the different types of paths that can be taken into the children of the parent node. As mentioned in the text, our traversal algorithm needs three stack items, here enumerated* I*,* II*, and* III*. Depending on which path (left, right, left then right, or right then left) a ray takes, the ray needs to be put into one or more stack items. For example, ray* R_a *hits the left child only, and is therefore put into* I*.* R_c *hits the right and then the left child, and is therefore put into* II *and* III*. Note that the order is important here, i.e., stack items are visited in the order given by the enumeration (*I*,* II*, and* III*).*

5. Traverse into right child, then left child.

We now need to represent these cases using stack items, each referring to the left child or right child. The first case means that the path is terminated, so it does not need a stack item. To accommodate the following two cases, it is enough to have one stack item for each node and the order of them is not important. However, in order to accommodate the two possible orders when hitting both children, we need to be able to order the left path before the right path in some cases, as well as order the right path before the left path in other cases. For this to work, three stack items are needed that reference the nodes left-right-left or right-left-right. This is explained in Figure 3 for left-right-left. Note that these cases can be accommodated using two different configurations. Which configuration is chosen is generally not important, however, the node that appears a single time may have sightly better SIMD efficiency because of the higher chance of maintaining larger packets as traversal continues.

As described previously, for efficient allocation of ray pointer lists to the different stack items, we pre-allocate one large list of ray pointers for each stack item that may be added during a traversal step. The pre-allocated ray pointer lists need to be able to hold all rays in the worst-case scenario. At most, each ray can exist in a single stack item at each level of the tree, so, if the maximum depth of the BVH is $d = 32$ and the number of rays traversed simultaneously is $m = 1024$, the maximum size of a pre-allocated ray pointer list is $m \times d = 32,768$.

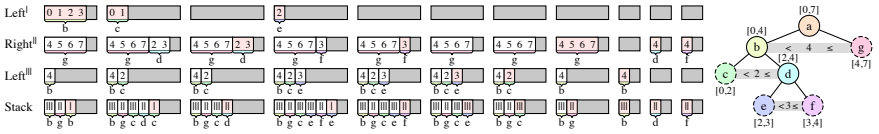


Figure 4: *Rays* $\{0,1,...,7\}$ *traversing the tree shown to the right. The nodes in the tree are named and leaf nodes are indicated using a dashed outline. For illustrative purpose, traversal is based on ray index, i.e., the interval within brackets indicates what rays will enter a node and the inequality between two siblings represents the ordering heuristic. No occlusion occurs in this example, so all paths are explored. The states of the pre-allocated ray pointer lists are shown from left to right as traversal progresses and the numbers within a range indicate individual rays, while the letter below indicates the associated node. At the bottom, the ray stream stack is shown, referencing ranges in the pre-allocated ray pointer lists. The range that is next to be processed is highlighted in light red. The initial state, to the far left, is the result of testing all rays against the immediate children of the root node. The strength of our algorithm is that all rays within a range can be efficiently processed using SIMD execution.*

A high level description of the traversal algorithm for BVH2 is shown below. In this case, the left-right-left configuration was chosen.

1. Build a ray stream of size m.
2. Allocate ray pointer lists $Left^{|}$, $Right^{||}$, and $Left^{|||}$, each with $m \times d$ entries, where d is the maximum depth of the BVH.
3. Initialize counters $cnt^{|}$, $cnt^{||}$, and $cnt^{|||}$ to 0.
4. Set *active rays* to be all rays in the ray stream.
5. Fetch the BVH root node, and make it *current node*.
6. Using SIMD, test all active rays against the current node:
   (a) If the node is internal, then test all rays against the BVs of the two children. The outcome determines what to do next:
      i. If the right child was hit, append a pointer to the ray to $Right^{||}$ at position $cnt^{||}$, and increment $cnt^{||}$.
      ii. If the left child was hit and it was closer than the right child, append to $Left^{|}$ and increment $cnt^{|}$.
      iii. If the left child was hit and it was farther than the right child, append to $Left^{|||}$ and increment $cnt^{|||}$.
   (b) If the node is a leaf, then test all rays against the triangles in the leaf.
7. Create stack items for all ray pointer lists where rays were added. Each item references the added range of ray pointers as well as the corresponding node (left or right). Then, push applicable stack items in the following order: $Left^{|||}$, $Right^{||}$, and $Left^{|}$, so that $Left^{|}$ is popped next.

8. If the stack is empty, we are done. Otherwise continue with the next step.
9. Pop a stack item and make the item's BVH node the current node. Also make the associated list of rays the active rays. To free up memory from the pre-allocated ray pointer lists, simply subtract the *cnt* variable this stack item refers to so that it points to the beginning of the current stack item. Added ray pointers will then overwrite the ones that were just read in the next traversal step.
10. Go to 4.

A detailed example of a few rays traversing a simple BVH2 is shown in Figure 4. Note how the ray stream stack consistently references the last range of rays in the pre-allocated ray pointer lists.

4.2 BVH4

In the case of a BVH with four children per internal node, i.e., a multi-BVH [24, 7, 5], the number of possible outcomes increases. In order to reduce that number, we restrict the ordering to that of two levels of a BVH2. Specifically, let $\{c_0, c_1, c_2, c_3\}$ be the children of an internal node of a BVH4. Conceptually, these nodes can then be put into a BVH2 where the first level has children $\{\text{left}, \text{right}\}$ and children$(\text{left}) = \{c_0, c_1\}$ and children$(\text{right}) = \{c_2, c_3\}$. This approximation allows us to accommodate all cases using 9 stack items, i.e., 3 were required by a BVH2 which indicates that 3×3 are sufficient for two levels in that tree.

Depending on how the BVH4 was created, this restriction can have little to no performance impact since the order flexibility is comparable to that of a BVH2. This would be the case if the BVH4 was built by collapsing nodes in a BVH2. If the BVH4 is instead built by iteratively splitting nodes with the best SAH cost, for example, care should be taken to group nodes that are in the same vicinity.

Analogous to Section 4.1, multiple configurations of stack items will be able to fulfill the ordering requirements. We used the following sequence of nodes for the stack items in our implementation: c_0-c_1-c_0-c_2-c_3-c_2-c_0-c_1-c_0.

For a BVH4, each ray may end up in three stack items at each level of the tree, in the worst case. However, looking at a single ray pointer list, it is sufficient to reserve space for all rays in all levels, since every stack item at the same depth level corresponds to a different ray pointer list. Therefore, each pre-allocated ray pointer list must have a size of $m \times d$, where m is the number of rays, and d is the maximum depth of the BVH, which is the same size required for BVH2 per list. The number of lists increase for BVH4, but on the other hand, the depth of the tree will decrease as well.

5 Implementation

As starting point for our implementation, we use Embree 2.0 [28], which is a collection of highly optimized ray traversal kernels and spatial data structures, and is widely used in the industry. The traversal interface of Embree was extended to include support for ray streams. The example path tracer [15] that comes with Embree, modified with a few performance improvements, was used to render all images, and it was also extended to support our algorithm. Most notably, in addition to the original recursive single-ray render loop, we added a separate render loop that builds ray streams, in order to trace many rays simultaneously. The new render loop eliminated recursion entirely and instead stored the required light path data for each ray, intersected all rays, and resumed each path in a loop. This approach is of similar spirit to what Laine et al. [17] did to separate material evaluation and path extension from ray casting in a GPU path tracer. Care was taken so that our new render loop traced the same number of rays and generated exactly the same end result as the original path tracer.

The path tracer is parallelized over multiple CPU cores by letting multiple threads work steal tiles of the final image. We made use of coherence within a tile by building a ray stream containing all primary rays generated from that tile. The initial tile size was 16×16 pixels and we used 16 samples per pixel, resulting in 4096 initial rays in a ray stream. Tile size variations are studied further in Section 6, however. Note that the number of rays in a ray stream decreases as light paths are terminated, so efficiency may be lowered after a few bounces. It could be possible to fill up the ray stream with rays from a neighboring tile, for example, but for simplicity and clarity of analysis, we did not attempt to do that.

Our traversal algorithm was optimized for the Intel Haswell architecture which provides support for 8-wide SIMD instructions for floating-point operations. The ray stream stores a single ray in two 32-byte structures, specifically, one used during BVH traversal and one used for triangle intersection. Each of these structures can be fetched using a single 256-bit AVX load. The BVH version of the ray holds the inverse of the ray direction to speed up slab tests. In addition, the ray origin is premultiplied with the inverse ray direction. Hit point information was also stored in a separate 32-byte structure, only accessed when an actual hit is registered.

When implementing the ray stream vs. internal BVH node test, different SIMD strategies were tested. The simplest and most parallel version was to test 8 rays at a time with Struct of Arrays (SoA) style SIMD. However, we found that the register pressure became too high in this case. The winning strategy turned out to be testing 2 rays at a time by running a 4-wide single-ray SIMD box-test over two rays simultaneously. Our ray pointer lists were implemented as arrays of 16-bit integers that index into the ray stream during traversal. For each loop iteration over the ray stream, we thus have to fetch two 16-bit indices and two 32-byte rays. Due to the frequent access, they are likely to be resident in the L1 or L2 cache for reasonable ray stream sizes. Appending ray indices to the ray pointer lists was done using scalar stores. We make use of fused multiply-adds to perform the

ray against bounding-box test, which is a high throughput, high latency operation. During this latency, we fetch the rays for the next loop iteration. In order to avoid fetching invalid rays for the next iteration, we always pad with valid ray indices at the end of each stack item, by duplicating the last ray index in the item. The details of our implementation for BVH2 is shown in Appendix A. While the tests for BVH2 and BVH4 are similar, the benefits of BVH4 are twofold: 1) ray loads are amortized over more bounding box tests, and 2) the capability to hide instruction latency increase due to an increased number of slab tests.

As an optimization, our algorithm switches to an optimized single-ray code path whenever there are less than n active rays. This code path is very similar to the single-ray code in Embree, but has been changed to work well with our 32-byte ray structures. For BVH2, n was set to 8, which is a fairly low number. For BVH4, n was instead set to 16, indicating that the overhead for this version is higher, which is expected since up to 9 stack items are added at each traversal step.

Another optimization we tried was to bin rays according to ray direction, which is a technique used by many before us (see Eisenacher et al.'s work [6] for a review of previous work). This was accomplished by creating 8 initial stack items in the ray stream stack, one for each possible sign configuration of x, y, and z. By sweeping over all rays once and appending each ray to the appropriate item, based on ray direction, we can ensure that all rays within a stack item share the same directional sign. We can then move any direction-dependent code out of the loop over the ray stream, which incidentally reduced register pressure too. Binning rays did not pay off for BVH2 as the lowered SIMD efficiency reduced performance, and the instructions eliminated due to constant ray sign partially resulted in more pipeline stalls. However, it was worthwhile for BVH4 as the register pressure in this case could be lowered by holding fewer node bounds at the same time. Furthermore, the BVH4 implementation has more parallel work, meaning that removed instructions gave a noticeable performance boost.

The renderer makes use of explicit light source sampling using shadow rays at each bounce, which warrants special treatment for these rays. When traversing a shadow ray against a BVH, the nearest hit point is of no interest. Instead, we just want to determine if anything is between a point and a light source. Because of this, it is no longer clear that traversing into the closest node is the best choice. Instead, traversal should continue into the child with the highest chance of occluding a shadow ray [14]. However, altering the BVH with a new cost metric would be detrimental to the comparison to previous algorithms in Embree, so they are therefore considered out of scope in this paper. Embree instead relaxes the ordering requirement for shadow rays to avoid some overhead. We do the same and implement shadow rays so that all rays in the ray stream take the same path in the tree, which decreases divergence and increases SIMD utilization. At each traversal step, we have a single stack item for each node for shadow rays. The order of these stack items are then determined by a simple heuristic: the stack item with most rays should be processed first. We have found this to work well in practice.

Embree also includes a set of packet traversal kernels [26], as well as hybrid ker-

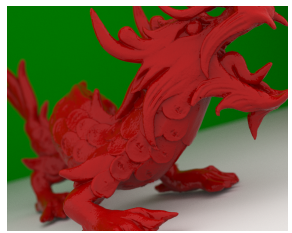
nels that starts with packets and switches to single-ray traversal as utilization becomes low [3]. To use these kernels, it is the responsibility of the renderer to manage ray packets. The example renderer that supports these kernels constitutes a total rewrite of the single-ray renderer using *ISPC* [20], which makes the entire renderer vectorized. This makes it a bit difficult to compare performance directly with our algorithm and single-ray traversal. For example, in the case of incoherent rays, shading and path extension get lower utilization as well. On the other hand, in the case of coherence between rays, shading would benefit from vectorization. We note this difference but still make comparisons in Section 6. Also note that with our algorithm, it may be possible to select between a scalar or vectorized version of a shader based on groups of rays created during traversal. However, we have chosen to keep the scalar shaders intact for simplicity and to make the comparison with single-ray traversal easier.

As ray-triangle test, all algorithms use parallel variations of the Möller-Trumbore test [18]. Our algorithm, as well as the single-ray test, make use of an 8-wide SoA SIMD implementation, intersecting 8 triangles against a single ray at a time. The packet traversal kernel instead tests 8 rays against 4 triangles at a time. The hybrid traversal kernel switches between 8 rays against 4 triangles and 1 ray against 4 triangles.

6 Results

We compare our algorithm to the single-ray, packet, and hybrid traversal kernels available in Embree 2.0. When possible, we compare using both BVH2 and BVH4 as spatial data structures, i.e., BVHs with two and four children, respectively. The BVH is built using the standard settings in Embree, which uses an SAH-binned construction algorithm [23]. Since our traversal algorithm was designed to generate exactly the same results as previous traversal algorithms, the generated images are exactly the same, and therefore, an image quality comparison is omitted. The algorithms were evaluated on an ultrabook laptop with a 4-core Intel Haswell Core i7-4750HQ CPU clocked at 2.0 GHz and with a turbo frequency of 3.2 GHz. This chip has 4×32 kB L1 instruction cache, 4×32 kB data cache, 4×256 kB L2 cache, 6 MB L3 cache, and 128 MB L4 EDRAM cache. The highest turbo frequency can only be reached when a single core is active and by monitoring the CPU frequency during rendering, we have found it to be stable at 2.8 GHz on all cores. The thermal design power (TDP) of the CPU and GPU combined is 47 W. Our modified version of Embree 2.0 was compiled with Intel C++ Studio XE 2013 Update 1 for Windows (compiler version 14.0.1) and all tests were performed in Windows 8. The packet renderer was compiled with ISPC version 1.6.0.

The evaluation is based on four test scenes, namely, CROWN, BENTLEY, DRAGON, and SANMIGUEL. The scenes and measured performance are shown in Table 1. As can be seen, our algorithm with BVH4 is the fastest for all scenes, while our BVH2 is the next to fastest. Of the competing traversal methods, there is no clear



	CROWN		BENTLEY		DRAGON		SANMIGUEL	
# triangles	4.9 million		2.3 million		7.3 million		7.9 million	
Resolution	1280 × 1024		1280 × 1024		1280 × 1024		1280 × 1024	
Ray queries[a]	61%		64%		64%		79%	
	Speed	**BW**	**Speed**	**BW**	**Speed**	**BW**	**Speed**	**BW**
Our BVH2	9.9 Mray/s	60 B/ray	15.7 Mray/s	40 B/ray	13.3 Mray/s	120 B/ray	7.8 Mray/s	130 B/ray
Our BVH4	**10.6** Mray/s	60 B/ray	**17.5** Mray/s	40 B/ray	**15.1** Mray/s	110 B/ray	**9.2** Mray/s	140 B/ray
Single BVH2	7.8 Mray/s	90 B/ray	11.7 Mray/s	80 B/ray	9.9 Mray/s	180 B/ray	5.6 Mray/s	150 B/ray
Single BVH4	*8.7* Mray/s	100 B/ray	12.8 Mray/s	70 B/ray	*11.9* Mray/s	120 B/ray	*6.7* Mray/s	180 B/ray
Packet BVH2	5.2 Mray/s	250 B/ray	8.3 Mray/s	100 B/ray	8.5 Mray/s	220 B/ray	4.0 Mray/s	290 B/ray
Packet BVH4	5.4 Mray/s	220 B/ray	8.9 Mray/s	120 B/ray	8.3 Mray/s	200 B/ray	4.0 Mray/s	330 B/ray
Hybrid BVH4	8.3 Mray/s	160 B/ray	*13.1* Mray/s	90 B/ray	11.5 Mray/s	170 B/ray	6.1 Mray/s	220 B/ray
Total | Traversal	22%	36%	34%	53%	27%	42%	37%	47%

[a]The fraction of rendering time spent on BVH traversal and triangle tests was measured using single-ray BVH4.

Table 1: *Performance (speed) numbers and bandwidth usage (BW), using a laptop CPU, for different algorithms when rendering four scenes. All images were generated by accumulating 16 samples per pixel each frame. The numbers include all work the renderer does, including generation of eye rays, shading, path extension, updating the frame buffer, and presenting the result on the screen. The overhead is approximately linear with respect to the number of rays per second, because higher performance means higher frame rate and more updates to the screen. The last row shows the total performance improvement to the left, and the speedup in terms of just the traversal (but including the additional overhead of the new render loop that builds ray streams for Our) to the right, for each scene. Those numbers are the ratios between our best method (Our BVH4) and the best competing technique (single, packet, hybrid), whose best result is marked in italics.*

winner: Hybrid BVH4 is best for BENTLEY, while Single BVH4 is best for the other. The last row reveals that total performance (including even frame buffer updates) is between 22–37% higher with our algorithm compared to the best of the competing methods. The last row also reveals if we only look at the time it takes to perform ray queries, i.e., including traversal and intersection testing (also including the additional overhead of the new render loop that builds ray streams), our traversal algorithm is 36–53% faster. Since our method is consistently and substantially faster for all test scenes, we believe that this is a very encouraging result.

Bandwidth to and from DRAM was measured in gigabytes per second using Intel VTune for an extended number of frames. Specifically, we let the renderer run for 30 seconds without measurements and then measured bandwidth during a 60 second interval. This number was divided by the number of rays per second because a higher throughput of rays would increase the bandwidth usage over a given time. The resulting value we use for comparison is thus bytes per ray, also shown in Table 1. Note that this is not the storage, but rather the average number of bytes needed to transfer to and from DRAM per ray. Our traversal methods use a lower amount of bandwidth per ray, or the same, as the other traversal algorithms. This is likely one of the sources to our performance improvement.

In Figure 5, we have evaluated the performance our algorithm with respect to different tile sizes. Recall that the number of samples per pixel have been fixed to 16, and then we vary the tile size, in order to build ray streams of different size. With a tile size of 12×12, the number of rays in a stream is $12 \times 12 \times 16 = 2304$, for example. As can be seen, 6400 initial rays in a ray stream performs the best, and this is equivalent to a tile size of 20×20 pixels. However, the performance is relatively stable for tile sizes between 16×16 and 24×24 pixels. The number of bytes per ray (B/ray), reported in Figure 5, has some variation among the scenes. Simply from our recursion-less render loop, one can expect a reduction in memory bandwidth because of better data locality. For example, ray queries is performed at the same time for all eye rays, and similarly for all rays after each bounce, which should increase locality of accesses in the BVH. Furthermore, shading happens without interference from traversal, which should improve texture caching. In addition, traversing large ray streams may also be beneficial for memory bandwidth if the reduction in node fetches happens to increase cache hits. This is, however, not certain, and the amount of cache lines the ray stream occupies in the cache may actually decrease cache hits for the BVH. From the plot we see that a reduction in memory bandwidth happens for BENTLEY and DRAGON, but not to a great extent for CROWN and SANMIGUEL, for the ray stream sizes that we investigated.

We also evaluated how our traversal algorithm behaved with varying field of view. The results can be seen in Figure 6, where it is clear that the advantage of our algorithm is relatively large for all angles, except for 8 degrees. In this particular case, only perfectly reflective surfaces are visible within the view, which makes all rays very coherent. As expected, Hybrid traversal performs extremely well in this case and wins by about 10%. The advantage that Hybrid has over our algorithm

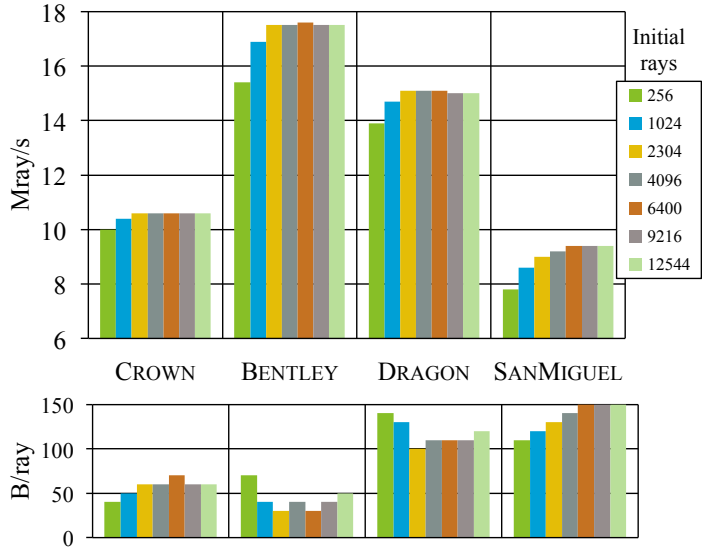


Figure 5: *Performance and bandwidth usage of Our BVH4 when changing the number of initial rays in a ray stream. In order to avoid changing the ray distribution, samples per pixel was fixed at 16, and tile size was varied from* 4×4 *to* 28×28 *pixels. The number of shadow rays that is traversed together is limited to the same number. However, since more shadow rays may be generated in the render loop (up to one per eye ray and light source), multiple batches of shadow rays may be traversed to test all of them.*

is twofold. First, it does not spend any time building ray streams and reordering rays, which is unnecessary overhead when rays are fully coherent. Second, the render loop and material evaluation is fully vectorized in this case due to the ISPC renderer and the same material covers the entire view. However, it is clear that this benefit disappears as soon as diffuse surfaces enters the view, and for 32 degrees, our total performance is about 43% higher compared to the fastest of Hybrid and Single BVH4. A nice property of our algorithm is that it does very well for various scenarios without any surprising performance cliffs. If anything, the diagram in Figure 6 reveals that the weaknesses of our algorithm is when all rays are coherent and they hit a reflective surface (so the rays continue to be rather coherent), or they hit a single large surface. However, it should be pointed out that our algorithm is rather fast in these cases, but the Hybrid may be faster.

In order to evaluate exactly where our performance improvement comes from, we measured the fraction of time spent in the ray query kernels (BVH traversal + intersection testing) using VTune during an extended period of time for our test scenes. The measured fraction was then multiplied by the ray throughput to get s/Gray (seconds per gigaray). The results are shown in Table 2. These numbers tell

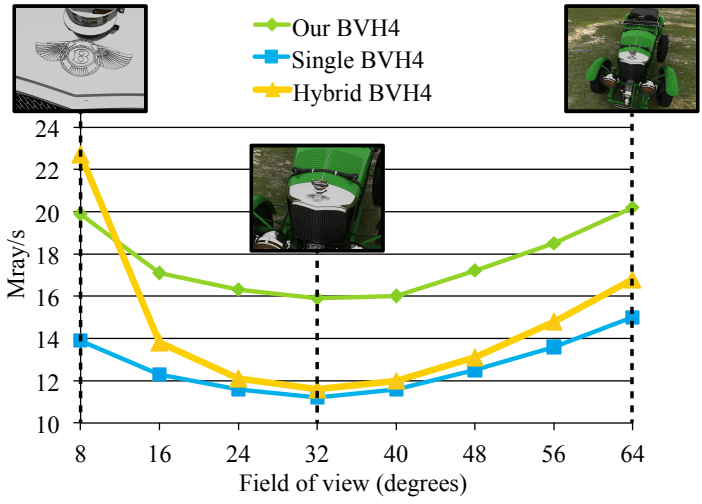


Figure 6: *In this evaluation, we changed the field of view from 8 to 64 degrees, which has similar effects as moving away from an object. As can be seen, our traversal algorithm is substantially faster for all but the smallest field of view. In the smallest field of view, only perfectly reflective surfaces can be seen which makes the rays very coherent. In this case, hybrid traversal does extremely well as expected, but quickly falls behind as diffuse surfaces enter the view. Also, when the field of view becomes very large, most of the image will contain a single large ground polygon, and therefore, the hybrid traversal slowly catches up with our performance.*

us exactly how much time ray queries take, and therefore, how much our traversal algorithm improves performance, disregarding the new render loop. This is very interesting since results including the render loop already were reported in Table 1. The cycles per instruction (CPI) shows how much the CPU pipeline stalls during traversal, and it is clear that our algorithm has low CPI. To highlight this fact, we show the same numbers (bottom row), but only for our optimized loop, which tests ray streams against the BVH. The rest of the ray query time in our algorithm is actually spent doing triangle intersection or single-ray traversal once coherence is low. Therefore, the optimized loop is the sole source for our increased traversal performance. The CPI value for that loop is very close to the optimal value for our target architecture, Haswell, which can issue two instructions per clock. So, its ideal CPI is 0.5, and the lower the reported CPI numbers are, the better.

For completeness, we also investigated the percentage of all ray vs. BVH node tests that was performed inside the optimized loop. We got the following results for BVH4 for the test scenes: CROWN: 76%, BENTLEY: 88%, DRAGON: 86%, and SANMIGUEL: 83%.

	CROWN		BENTLEY		DRAGON		SANMIGUEL	
	CPI	Time per ray	CPI	Time per ray	CPI	Time per ray	CPI	Time per ray
Single BVH4 Traversal	1.40	70 s/Gray	1.28	50 s/Gray	1.29	54 s/Gray	1.45	117 s/Gray
Our BVH4 Traversal	1.02	54 s/Gray	0.88	33 s/Gray	0.91	39 s/Gray	0.95	81 s/Gray
Our core loop only	**0.61**	16 s/Gray	**0.61**	13 s/Gray	**0.60**	15 s/Gray	**0.60**	31 s/Gray

Table 2: *Here, we measure the fraction of time spent in the traversal kernels per ray (in seconds per gigaray). We compare Our BVH4 against Single BVH4. In addition, we present cycles per instruction (CPI) as a measurement of how much the CPU pipeline stalls during traversal. At the bottom, we show the same measurements, but only for our optimized loop that tests a ray stream against a BVH4.*

7 Conclusions and Future Work

Over the last 10–15 years, a lot of optimization research effort has been spent on building better BVHs faster, and on parallelizing and SIMDifying the core of the ray tracing algorithm, namely, traversal, intersection testing, and shading. Embree is one of the most optimized ray tracing frameworks, for a variety of CPUs, that we know of, and it is widely used in the industry for this reason. By completely changing the interface to Embree, we have been able to implement our novel algorithm that traverses the BVH with large ray streams using a dynamic descent in the top part of the BVH, and then switches to single-ray traversal. As we have demonstrated, when measuring only the time it takes to perform the traversal, our new algorithm is up to 53% faster. For future work, we would like to test whether a memory-mapped layout of the scene can be used to automatically render huge scenes with our traversal method. We believe this should be possible, since our algorithm is good at reducing memory bandwidth usage by visiting the same node with many rays. Furthermore, we would like to investigate whether our method is at all feasible on a GPU and for a Xeon Phi accelerator.

Acknowledgements

Thanks to Tom Piazza, David Blythe, and the whole Advanced Rendering Technology group, all at Intel. Some of Rasmus' work was done during an internship at Intel, San Francisco, hosted by Charles Lingle (thanks!). Tomas is a *Royal Swedish Academy of Sciences Research Fellow*, supported by a grant from the Knut and Alice Wallenberg Foundation. Thanks to Jim Nilsson, Ingo Wald, and Sven Woop for ray tracing discussions and proofreading, and in particular, thanks to Carsten Benthin and Jacob Munkberg for lots of help. Thanks to Martin Lubich for the Crown model (www.loramel.net).

Bibliography

[1] ÁFRA, A. T., AND SZIRMAY-KALOS, L. Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Computer Graphics Forum 33*, 1 (2013), 129–140.

[2] BARRINGER, R., AND AKENINE-MÖLLER, T. Dynamic Stackless Binary Tree Traversal. *Journal of Computer Graphics Techniques 2*, 1 (March 2013), 38–49.

[3] BENTHIN, C., WALD, I., WOOP, S., ERNST, M., AND MARK, W. Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 9 (2012), 1438–1448.

[4] BOULOS, S., WALD, I., AND BENTHIN, C. Adaptive Ray Packet Reordering. In *Symposium on Interactive Ray Tracing* (2008), pp. 131–138.

[5] DAMMERTZ, H., HANIKA, J., AND KELLER, A. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum 27*, 4 (2008), 1225–1233.

[6] EISENACHER, C., NICHOLS, G., SELLE, A., AND BURLEY, B. Sorted Deferred Shading for Production Path Tracing. *Computer Graphics Forum 32*, 4 (2013), 125–132.

[7] ERNST, M., AND GREINER, G. Multi Bounding Volume Hierarchies. In *IEEE Interactive Ray Tracing* (2008), pp. 35–40.

[8] GARANZHA, K., AND LOOP, C. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298.

[9] GRIBBLE, C. P., AND RAMANI, K. Coherent Ray Tracing via Stream Filtering. In *Symposium on Interactive Ray Tracing* (2008), pp. 59–66.

[10] HANRAHAN, P. Using Caches and Breadth-First Search to Speed Up Ray Tracing. In *Graphics Interface* (1986), pp. 56–61.

[11] HAPALA, M., DAVIDOVIC, T., WALD, I., HAVRAN, V., AND SLUSALLEK, P. Efficient Stack-less BVH Traversal for Ray Tracing. In *27th Spring Conference of Computer Graphics* (2011), pp. 29–34.

[12] HENNESSEY, J. L., AND PATTERSSON, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[13] HUGHES, D. M., AND LIM, I. S. Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1555–1562.

[14] IZE, T., AND HANSEN, C. RTSAH Traversal Order for Occlusion Rays. *Computer Graphics Forum 30*, 2 (2011), 297–305.

[15] KAJIYA, J. T. The Rendering Equation. *Computer Graphics (Proceedings of ACM SIGGRAPH 86) 20*, 4 (1986), 143–150.

[16] LAINE, S. Restart Trail for Stackless BVH Traversal. In *High Performance Graphics* (2010), pp. 107–111.

[17] LAINE, S., KARRAS, T., AND AILA, T. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *High Performance Graphics* (2013), pp. 137–143.

[18] MÖLLER, T., AND TRUMBORE, B. Fast, Minimum Storage Ray-triangle Intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

[19] MORA, B. Naive Ray-tracing: A Divide-and-conquer Approach. *ACM Transactions on Graphics 30*, 5 (2011), 117:1–117:12.

[20] PHARR, M., AND MARK, W. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing (InPar)* (2012), pp. 1–13.

[21] RAMANI, K., GRIBBLE, C. P., AND DAVIS, A. StreamRay: A Stream Filtering Architecture for Coherent Ray Tracing. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 325–336.

[22] TSAKOK, J. A. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *High Performance Graphics* (2009), pp. 151–158.

[23] WALD, I. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing* (2007), pp. 33–40.

[24] WALD, I., BENTHIN, C., AND BOULOS, S. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *IEEE Symposium on Interactive Ray Tracing* (2008), pp. 49–57.

[25] WALD, I., GRIBBLE, C. P., BOULOS, S., AND KENSLER, A. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Tech. Rep. UUSCI-2007-012, 2007.

[26] WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164.

[27] WHITTED, T. An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (1980), 343–349.

[28] WOOP, S., FENG, L., WALD, I., AND BENTHIN, C. Embree Ray Tracing Kernels for CPUs and the Xeon Phi Architecture. In *ACM SIGGRAPH 2013 Talks* (2013).

A Core Loop Implementation for BVH2

Here we list our core loop implementation for testing a ray stream against an internal node of a BVH2 acceleration structure. The code is mainly written using AVX2 intrinsic functions, but we present it using a simplified syntax to improve readability.[1] The loop tests two rays against the two bounding boxes of the children and stores ray indices into the pre-allocated ray pointer lists. The input bounds are organized as $[min_l, min_r, max_l, max_r]$, i.e., a 4-wide vector, for each coordinate axis, and are broadcast to 256 bits by duplicating the 4 floats to both 128-bit lanes. After running through the loop, stack items are created for the ray pointer lists where rays actually were added. That part has been omitted for brevity.

```
void coreLoopBvh2(
  // AABBs for left and right children.
  m256 bbX, m256 bbY, m256 bbZ,
  // Pointers to the end of the pre-allocated ray pointer lists.
  uint16* list0, uint16* list1, uint16* list2,
  // Active rays (pointing into one of the pre-allocated ray pointer lists).
  uint16* activeRays, uint16* lastActiveRay)
  {
  m256 negMask = set1(-0.0f);
  m256 posNegMask = setr(0.0f, 0.0f, -0.0f, -0.0f, 0.0f, 0.0f, -0.0f, -0.0f);

  m256 bbXneg = shuffle(bbX, bbX, M(1,0,3,2));
  m256 bbYneg = shuffle(bbY, bbY, M(1,0,3,2));
  m256 bbZneg = shuffle(bbZ, bbZ, M(1,0,3,2));

  bbX ^= posNegMask; bbY ^= posNegMask; bbZ ^= posNegMask;
  bbXneg ^= posNegMask; bbYneg ^= posNegMask; bbZneg ^= posNegMask;

  uint32 nextIndexA = activeRays[0];
  uint32 nextIndexB = activeRays[1];

  m256 dirNearOrgFarA = load(rayData + nextIndexA);
  m256 dirNearOrgFarB = load(rayData + nextIndexB);

  m256 dirNear = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 0|(2<<4));
  m256 orgFar = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 1|(3<<4));

  m256 orgFarNeg = orgFar ^ negMask;

  m256 dirX = shuffle(dirNear, dirNear, M(0,0,0,0));
  m256 dirY = shuffle(dirNear, dirNear, M(1,1,1,1));
  m256 dirZ = shuffle(dirNear, dirNear, M(2,2,2,2));

  uint32 mask = 0;
  uint32 indexA = 0, indexB = 0;

  for (; activeRays < lastActiveRay; ) {
    uint32 leftHit = mask >> 4;
    uint32 rightHit = (mask >> 5) & 1;
    uint32 order = (mask >> 6) & rightHit;

    m256 bbXray = blendv(bbX, bbXneg, dirX);
    m256 orgX = shuffle(orgFar, orgFarNeg, M(0,0,0,0));
    m256 bbYray = blendv(bbY, bbYneg, dirY);
    m256 orgY = shuffle(orgFar, orgFarNeg, M(1,1,1,1));
    m256 bbZray = blendv(bbZ, bbZneg, dirZ);
    m256 orgZ = shuffle(orgFar, orgFarNeg, M(2,2,2,2));
```

[1] To restore the original source, add `_mm256_` before and `_ps` after functions, replace `m256` with `__m256`, replace `M` with `_MM_SHUFFLE`, and replace the $\wedge$ operator with a call to `_mm256_xor_ps` when used with vector registers.

```
    *list0 = indexB; *list1 = indexB; *list2 = indexB;
    list0 += leftHit & (1^order);
    list1 += rightHit;
    list2 += leftHit & order;

    indexA = nextIndexA;
    indexB = nextIndexB;
    activeRays += 2;
    nextIndexA = activeRays[0];
    nextIndexB = activeRays[1];

    m256 nearFarX = fmsub(bbXray, dirX, orgX);
    m256 nearFarY = fmsub(bbYray, dirY, orgY);
    m256 nearFarZ = fmsub(bbZray, dirZ, orgZ);
    m256 nearFarRay = shuffle(dirNear, orgFarNeg, M(3,3,3,3));

    dirNearOrgFarA = load(rayData + nextIndexA);
    dirNearOrgFarB = load(rayData + nextIndexB);

    m256 nearFar = max(max(nearFarRay, nearFarX), max(nearFarY, nearFarZ));

    mask = movemask(cmple(shuffle(nearFar, nearFar, M(0,1,1,0)),
                          shuffle(nearFar ^ negMask, nearFar, M(0,0,3,2))));

    dirNear = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 0|(2<<4));
    orgFar = permute2f128(dirNearOrgFarA, dirNearOrgFarB, 1|((3<<4));

    dirX = shuffle(dirNear, dirNear, M(0,0,0,0));
    dirY = shuffle(dirNear, dirNear, M(1,1,1,1));
    dirZ = shuffle(dirNear, dirNear, M(2,2,2,2));

    orgFarNeg = orgFar ^ negMask;

    leftHit = mask;
    rightHit = (mask >> 1) & 1;
    order = (mask >> 2) & rightHit;

    *list0 = indexA; *list1 = indexA; *list2 = indexA;
    list0 += leftHit & (1^order);
    list1 += rightHit;
    list2 += leftHit & order;
  }

  if (indexA != indexB) { // Checks if the last index was duplicated (padding).
    uint32 leftHit = mask >> 4;
    uint32 rightHit = (mask >> 5) & 1;
    uint32 order = (mask >> 6) & rightHit;

    *list0 = indexB; *list1 = indexB; *list2 = indexB;
    list0 += leftHit & (1^order);
    list1 += rightHit;
    list2 += leftHit & order;
  }
}
```