Coherent Ray Tracing for Complex Light Transport Effects

Ryan S. Overbeck

Advised by: Ravi Ramamoorthi

Submitted in partial fulfillment of the

Requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2010

ABSTRACT

Coherent Ray Tracing for Complex Light Transport Effects

Ryan S. Overbeck

With current rendering algorithms, it is now possible to generate photorealistic images on a single desktop computer. At the core of many of these algorithms is a ray tracer, which is used to simulate physical light transport. All too often, the ray tracer is also the performance bottleneck, and as a result, photorealistic images often take hours to generate and/or employ approximations which detract from realism. Interactive content, by comparison, often appears dull and artificial.

The past decade has seen vast improvements to the core ray tracing algorithms, promising greater realism with faster render times. A large body of work has been dedicated to so-called "coherent ray tracing" methods, where the coherence between neighboring rays is exploited to improve ray tracing performance. These algorithms map cleanly to modern processors' Single Instruction Multiple Data (SIMD) units, render primary visibility and point-light shadows at interactive to real-time rates, and provide over an order of magnitude performance improvement over traditional one-at-a-time ray tracers in some cases. These benefits do not come cheaply: we must sacrifice some of the generality of one-at-a-time ray tracing. Moreover, coherence is not guaranteed for more complex light transport, beyond primary visibility and point-light shadows. So it remains unclear as to how well these new methods extend to the more complex light transport effects, such as area lighting, reflections, refractions, depth of field, motion blur, and global illumination.

In this thesis, I propose three bodies of work which introduce and evaluate new algo-

rithms for coherent ray tracing dedicated to accelerating complex light transport effects: a real-time beam tracer with application to exact soft shadows from area light sources, large ray packets for real-time Whitted ray tracing, and adaptive wavelet rendering for general high-dimensional effects.

I first introduce a highly optimized beam tracer which computes noise-free soft shadows in seconds and renders antialiased primary visibility and point-light shadows in real-time. Whereas bounding frusta have previously been used to cull away expensive per ray intersection tests for entire ray packets, I use the frustum as the atomic ray primitive. This results in faster visibility testing when the scene geometry is coherent, and also provides an exact visibility solution for efficient antialiasing and analytic soft shadows.

Then, to handle reflections and refractions, I construct an interactive Whitted ray tracer composed of new algorithms for large ray packets and frustum culling. Within this framework, I offer a thorough analysis of several coherent ray tracing algorithms, and observe strong benefits, albeit less than for primary visibility and point-light shadow rays. Even in situations of extreme incoherence, large ray packets tend to be 3×–6× faster than 4-wide SIMD rays.

Finally, I propose adaptive wavelet rendering for general high-dimensional effects, such as area lighting, depth of field, motion blur, and diffuse inter-reflections. This is an adaptive Monte Carlo algorithm, but rather than adapt to a per pixel measure of variance, this new algorithm adapts to variance in a multi-scale wavelet basis. Thus it adapts to smooth sources of variance, such as the blur from an out-of-focus camera, by sampling at an effectively lower resolution, while targeting edges with more focused sample distributions. The remaining fine-scale noise is removed by a novel wavelet reconstruction filter. One notable aspect of adaptive wavelet rendering is that it adaptively samples image regions rather than points, and so is well suited to coherent ray tracing techniques. This new algorithm often achieves near-reference quality images with general combinations of high-dimensional effects with an average of only 32 samples per pixel, far fewer than required by traditional means. Moreover,

the algorithm is efficient, and maintains low overhead even when used with an optimized coherent ray tracer.

Together, these three works improve coherent ray tracing performance for a broad range of complex light transport effects that are vital for photorealistic rendering. They increase the quality of interactive content and decrease the render times of offline photorealistic content, bringing us two steps closer to interactive photorealistic images.

Contents

List of Figures

List of Tables

Acknowledgements

I consider myself lucky to have had Prof. Ravi Ramamoorthi as my thesis advisor. He pushes hard, but the only agenda he ever pushed on me was to follow my own ideas and interests with intensity and perseverence. He treats everybody's ideas as valuable, even when they don't align with his own. I look forward to an opportunity to return the favor.

The other students, professors, and staff of the Columbia Vision and Graphics Center have provided invaluable assistance. Together, these professors work hard to create an environment of collaboration: Ravi Ramamoorthi, Eitan Grinspun, Shree Nayar, Peter Belhumeur, Steve Feiner, Tony Jabara, John Kender, and Peter Allen. The attitude between the students is one of support and cooperation with no evidence of competition. The success of one is considered the success of all, and it is common for one student to set aside their own work in order to help another get through a particularly difficult milestone. I worked closely with Aner Ben-Artzi, Craig Donner, Kevin Egan, Jin-Wei Gu, Bo Sun, Simon Premoze, Yu-Ting Tseng, Sebastian Enrique, Akash Garg, Dhruv Mahajan, Kalyan Sunkavalli, David Harmon, Etienne Vouga, Miklos Bergou, Rony Goldenthal, Kshitiz Garg, Eric Risser, Oliver Cossairt, and Saurabh Mathur within this positive atmosphere. I wish them all the best in their future endeavors and hope to work with every one of them again. Out of the administrative staff, Anne Fleming and Lily Secora truly go above and beyond in their duties, and deserve to be noted above all others.

I have received considerable aid from the research community outside of Columbia University. I owe Prof. Hanspeter Pfister a special debt of gratitude: I never intended to

attend graduate school until he encouraged me to do so. His enthusiasm for graphics is too infectious to say 'no' to.

I thank Intel Corporation and their graphics research group for supporting my research. Dr. William Mark has been an excellent source of support, and his voice-of-reason has helped shape a significant portion of my work. Both he and Dr. Alexander Reshetov continue to support me in reaching my professional goals.

Finally, I owe my deepest gratitude to my family and friends who make life beautiful and worth living. A special thanks to my wife, Londa, for her support, love, and (most of all) patience.

This thesis is dedicated to my father, Dr. James Wilcox Overbeck, 1939–2009. Some of my most precious memories are of the two of us driving to my peewee hockey games, with him quizzing me with word problems.

Chapter 1

Introduction

Non-interactive computer generated imagery is now often indistinguishable from photographs. This is largely due to our ability to efficiently simulate the complex light transport effects which affect appearance. Ray tracing is the most general algorithm for accurately simulating geometric light transport, and it is at the heart of many physically based rendering solutions, such as Whitted ray tracing, path tracing, distributed ray tracing, and photon mapping. Unfortunately, ray tracing is often slow, and the time to generate photorealistic images may be prohibitive, taking hours to render a single image on a commodity workstation. This greatly limits our experience with photorealistic imagery. Interactive applications, such as video games, are still artificial in appearance, and even for offline production images for feature films, the design process is hampered by the artist's inability to freely iterate and improve their work.

In an effort to bring interactive content closer to photorealism, researchers have recently introduced vast improvements to the core ray tracing algorithms. At the heart of this recent focus, are so-called *coherent ray tracing* algorithms. Using these methods, it is now possible to ray trace simple scenes with primary visibility and point-light shadows in real-time. However, photorealism requires more complex light transport than primary visibility and point-light shadows. Real-world scenes have soft shadows from area light sources, glossy

reflections and refractions, blurry depth of field effects from out-of-focus camera lenses, motion blur, and diffuse inter-reflections.

In this thesis, I seek to leverage coherent ray tracing techniques to accelerate complex light transport effects. I offer three pieces of work to achieve this goal: a real-time beam tracer with application to exact soft shadows from area light sources; large ray packets for real-time Whitted ray tracing; and adaptive wavelet rendering for general high-dimensional effects. Together, these three works address nearly the full gamut of light transport effects required for photorealistic rendering.

1.1 Coherent Ray Tracing

In its most basic form, ray tracing is an exceedingly simple and general algorithm. It is essentially a database query, where the input query is a ray of light, the database is a 3D scene, and for output, we seek the intersection between the ray and the scene. To make this query efficient, the geometric primitives, which constitute the scene, are organized into an acceleration structure. This structure usually consists of a hierarchy of simpler geometry, such as axis-aligned bounding boxes. Traditionally, rays are traced one-at-a-time, each one traversing the acceleration structure and individually tested against the geometry primitives.

Coherent ray tracing techniques further accelerate a ray tracer by answering many ray queries at the same time. Most rendering solutions require hundreds of millions of rays to compute an image. Many of these rays will travel close together in 3D space, visiting the same nodes in the acceleration structure, and often even intersecting the same geometry primitives. *Ray coherence* refers to this quality of multiple rays traversing similar paths through the scene, and it provides an opportunity to speed up a ray tracer.

There are many ways that coherence can accelerate the ray tracing process. Algorithmic amortization is one. Some operations are expensive when performed individually for each ray query, but when amortized across many queries, they become efficient. For example,

loading geometric primitives into memory can be expensive when executed independently for every ray. However, if one primitive needs to be tested for intersection against many rays, it should be loaded only once for all rays. In some other cases, it may be possible to perform one conservative intersection test for an entire group of rays in order to avoid many tests for each individual ray.

Perhaps more importantly, coherent ray tracing methods expose data-parallelism, which allows a ray tracer to utilize the Single Instruction Multiple Data (SIMD) arithmetic units on modern CPUs. SIMD units perform the same operation on multiple data for the cost of one. Most general function CPUs currently offer 4-wide SIMD units, and future hardware [59] increases this to 16-wide, indicating a trend to wider and wider SIMD. On a 16-wide unit, a perfectly data-parallel algorithm can theoretically achieve a 16× performance boost. In reality, data-parallelism is difficult to exploit, and practical performance benefits tend to be significantly less. In fact, even the most data-parallel algorithms rarely achieve more than $\frac{3}{4}$ of the theoretical maximum. Therefore, algorithmic data-parallelism is a precious commodity.

While traditional one-at-a-time ray tracers are ill-suited to SIMD computation, coherent ray tracers are innately data-parallel. Since many ray queries need to be solved simultaneously, they can be mapped to the multiple SIMD channels. For primary visibility and point-light shadows, coherent ray tracers tend to achieve a 3× performance boost from 4-wide SIMD [76]. Even for less coherent rays, and larger SIMD widths, recent research [24, 10] suggests that it is possible for coherent ray tracing to maintain %50–%90 SIMD utilization. These results indicate that coherent ray tracing may be vital to the future of ray tracing, in order to efficiently map to future hardware.

Driven by the need to convert this potential to actual performance, recent research offers a multitude of coherent ray tracing algorithms. Many of these algorithms are based on ray packets (also called breadth-first ray tracing or ray stream tracing) [76, 72, 9, 74, 24], where many rays are cast at the same time. Ray packets can amortize computation costs for

scene traversal and shading across all rays in the packet. Moreover, on SIMD architectures, multiple rays can be tested at the cost of a single ray. Another approach is to use bounding frusta or interval arithmetic [57, 11, 74] around ray packets to cull away many expensive ray–geometry intersection tests. For primary and point-light shadow rays, packet ray tracing with frustum culling often performs over an order of magnitude faster than one-at-a-time ray tracing.

Despite the proven benefits of coherent ray tracing algorithms, there are several hurdles that prohibit their broader adoption. Many traditional ray tracing algorithms and rendering APIs were designed specifically for one-at-a-time ray tracing, and do not trivially translate to ray packets and breadth-first processing. As a result, incorporating coherent ray tracing into existing renderers can be a lengthy and costly task. The benefits must be well proven and the algorithms mature before they become worth the expense. Coherent ray tracing algorithms must demonstrate their benefits on production quality scenes and images.

This thesis offers both new algorithms and a thorough evaluation of old algorithms to demonstrate coherent ray tracing on the complex light transport effects required for production quality rendering. Prior to this work, research focused almost exclusively on primary visibility and point-light shadows. The rays required for computing the more complex light transport effects are significantly less coherent, and therefore less apparently amenable to existing coherent ray tracing methods. To move forward, we require new algorithms that expose and exploit ray coherence. It is also necessary to re-evaluate old algorithms under the duress of incoherence.

1.2 Thesis Overview

In this thesis, I offer three pieces of work which utilize coherent ray tracing for efficient, high-quality rendering of complex secondary effects. All works demonstrate large performance gains over traditional one-at-a-time ray tracing algorithms, and in many cases, final render

quality is improved.

First, in Chapter 2, I introduce a real-time beam tracer with application to exact soft shadows, which was originally published and presented in [51]. This work takes ray coherence to the logical extreme. Rather than using a ray packet with a bounding frustum, I propose to use only the frustum, and directly split it against the scene geometry. This beam tracer is designed for 4-wide SIMD units and uses new algorithms for kd-tree traversal and triangle intersection. It is dramatically faster than previous beam tracing implementations. In fact, in many cases where the beam's volume is small relative to the scene's geometry, it even out-performs the fastest known packet ray tracer. Beyond pure wall-clock performance, beams offer qualitative benefits over rays such as improved antialiasing. I also demonstrate a new application for beam tracing: computing exact noise-free soft shadows.

In Chapter 3, I present a real-time Whitted ray tracer using large ray packets and frustum culling. This work was published and presented in [52]. Beyond just primary visibility and point-light shadows which are known to be fast using coherent ray tracing, a Whitted ray-tracer also requires ideal reflections and refractions. These rays tend to be significantly less coherent, so I introduce partition traversal, a new ray packet algorithm which is robust to degradation in coherence. I also introduce a new approach to construct bounding frusta around general ray types including reflection and refraction rays. I then evaluate these and other algorithms within an interactive Whitted ray tracer, and study how well these algorithms respond to varying ray packet size, geometric complexity, and ray recursion complexity. I find that packet ray tracing still offers significant benefits even for these less coherent rays, but frustum culling's impact is somewhat reduced.

Finally, in Chapter 4, I focus on general high-dimensional effects, including depth of field, motion blur, area lighting, antialiasing, and diffuse inter-reflections. All of these effects can be evaluated simultaneously by computing a high-dimensional Monte Carlo integral, but standard Monte Carlo approaches are prone to severe noise. I therefore propose *adaptive wavelet rendering* to adapt to image space variance and remove noise. This work

will be published and presented in [50] after this thesis. In contrast to traditional methods which adapt to a per-pixel measure of variance, this new algorithm adapts to variance in a multi-scale wavelet basis, and distributes samples to reduce variance at specific scale coefficients. To remove the noise in under-sampled regions, I introduce a novel wavelet reconstruction which effectively chooses the smoothest image that fits the Monte Carlo samples. In regions of smooth variance, I sample at an effectively lower resolution, and the reconstruction removes the fine-scale noise. Near edges, samples are focused at the fine-scale coefficients where the edges are resolved more clearly. This new algorithm is particularly well-suited for acceleration by coherent ray tracing techniques because it adapts to image regions rather than individual pixels, and it introduces significantly less overhead than other adaptive algorithms.

These three works are significantly distinct from each other, both with respect to the rendering effects they seek to accelerate as well as the approaches they use to exploit coherence. Their difference from each other highlights the broad potential for future research in coherent ray tracing. In Chapter 5, I offer closing remarks and address some of the directions for future research indicated by this thesis.

Chapter 2

Beam Tracing for Efficient Antialiasing and Analytic Soft Shadows

Soft shadows are valuable for photorealistic rendering. They provide visual depth cues and help to set the mood in an image. However, generating accurate soft shadows involves solving an expensive integral at each pixel.

The most common technique to approximate such integrals in computer graphics is Monte-Carlo integration using distributed ray tracing. These methods sample the light source many times and are prone to noise due to variance in the estimate. While importance sampling and light source stratification can reduce this noise, it remains, and increasing the sampling density provides only diminishing returns [60].

Ray tracing has a long history in graphics [78], and has received particularly focused attention in the past few years spurred by new acceleration techniques which determine primary visibility for complex scenes in real-time. These methods use ray bundles [76], frustum proxies for kd-tree traversal [57, 74, 68, 32], and make efficient use of the register SIMD (e.g., SSE) instructions available on most modern processors [76, 57, 74]. All of these approaches leverage geometric coherence of neighboring rays as they traverse a scene. However, per-pixel shading can still significantly reduce their performance (even by as much

as 33% or more [57]). Furthermore, initial indications are that the benefits of these methods are relatively modest for secondary effects such as soft shadows. The recent results from Boulos et al. [9] show only a 2×–3× performance improvement for ray packets as compared to individual rays for distribution ray tracing tasks.

When taken to the limit, full use of geometric coherence leads us to beam tracing. Beam tracing was introduced by Heckbert and Hanrahan [33] as a method of leveraging the geometric coherence of groups of rays by tracing a volume of rays instead of each ray individually. While not as general a rendering solution as ray tracing, beam tracing can solve many problems including antialiasing, specular reflections, and soft shadows. However, it has received limited attention from the rendering community since its inception for two reasons. First, the basic geometry intersection tests become significantly more complicated when moving from rays to beams. Second, there has not been significant success in applying acceleration structures to beam tracing.

In this chapter, I develop novel acceleration and intersection techniques for beam tracing. I can obtain real-time results for primary rays, faster than the best accelerated ray-tracing methods [57] for many scenes where the average visible triangle size is large compared to the sample density. Moreover, it is important to note a fundamental difference between beam and ray tracing. For rendering primary visibility, ray tracing first point samples the image, and then determines visibility for each sample. On the other hand, beam tracing deals directly with coherent area elements (visible triangles), and delays image space sampling to the very end. In my beam tracing system, this final image-space sampling is delegated to the highly parallel and specialized GPU rasterizer. Therefore, beam tracing can retain coherence much further, which we can exploit for antialiasing, shading, and point light shadows. Perhaps more importantly, beam tracing is particularly well suited to secondary visibility for area lighting. No point sampling of the light is needed in this case—only area samples are required. As seen in Fig. 2.11, I can obtain essentially *exact* soft shadows from area lights significantly faster than previously possible. The shadows are *exact* in the

sense that they are computed using an exact representation of the visibility between the light source and the shade point. Compared to a ray tracer which approximates soft shadows using 256 light source samples, this beam tracer can be 10×–40× faster.

The performance of this method derives from two main technical contributions. I have designed a new algorithm for beam–triangle intersection (Sec. 2.2.2) which splits beams at triangle edges. This algorithm combines the successes of fast ray–triangle intersection and polygon clipping algorithms. The computational cost of the beam splitting operations is nearly equivalent to generating only one new ray in a conventional ray tracer. I also introduce the first effective method of kd-tree traversal (Sec. 2.2.3) for beam tracing. While frusta have previously been used to determine a conservative traversal estimate for a bundle of rays (and my approach is inspired by these works) I found that beam tracing can benefit even more than rays from efficient kd-traversal. As a high level decision, I specifically designed both my beam–triangle intersection and kd-tree traversal algorithms to be parallelizable through use of SIMD SSE instructions. For beam tracing, this data parallel design leads to new algorithms quite distinct from their serial counterparts.

The beam tracer's performance now scales linearly with the number of *visible* triangles with relatively minimal dependence on absolute scene size. Compared to ray tracing for primary visibility (Sec. 2.3), I achieve a speed-up relative to the ratio of ray sample density to the average visible triangle surface area. When the triangles are large relative to sample density, beam tracing can be more than an order of magnitude faster than the fastest ray tracers, and it is competitive even for moderate to large scenes (thousands of visible triangles).

However, perhaps the biggest advantage of beam tracing is for precise soft shadows from compact area light sources. As my analysis in Sec. 2.5 shows, the average number of visible triangles (and hence hit beams) at each pixel tends to be significantly lower than for primary visibility (less than 10 for my test scenes), enabling substantial performance improvements over ray tracing. Even in highly tesselated scenes, where the performance

benefits relative to ray tracing are reduced, this beam tracer achieves exact noise-free soft shadows, independent of resolution in a matter of seconds.

2.1 Previous Work

I first discuss previous work on high quality soft shadows. I then consider methods for acceleration of primary rays in ray tracing, and early efforts at beam tracing.

2.1.1 Accurate Soft Shadows

Sampling: The sampling-based approach was first introduced as distributed ray tracing [15]. Using Monte Carlo integration, it solves a variety of rendering problems including motion blur, depth of field, fuzzy reflections, and soft shadows. As with any sampling based method, variance in the estimate is evidenced by noise in the result (see Figure 2.11). In order to reduce variance to a reasonable tolerance, a large number of rays are often required—typically 256 shadow rays for generating a single 24-bit image, and 1024 or more shadow rays for production level rendering.

There are also methods that use visibility coherence to reduce the number of shadow rays [1]. However, they typically offer only a 3×–4× speedup for intricate shadowing. Moreover, they introduce measurable overhead when used with a heavily optimized ray tracer, such as the one I use in my comparisons in Section 2.5, which can significantly reduce the benefit.

Exact Methods: Only an exact solution is capable of guaranteeing an accurate and noise-free result and many are available. These methods gather exact occluder geometry and integrate over the resulting area elements. Hart et al. [29] turn point samples into area samples using a flood-fill algorithm. Most methods search for silhouette edges through back-projection and/or tracking visibility events [65, 17]. While these methods can produce

very high quality results, they tend to be much slower than the sampling approaches and scale very poorly with scene geometry. My beam-tracing approach also uses an exact representation of the occluding geometry for noise-free images, but is sensitive only to visible scene complexity making it fast even for large scenes.

Soft Shadow Volumes: The most recent work, introduced by Laine et al. [40], uses a mixture of the visibility event and sampling approaches. Building upon the idea of shadow volumes [16], they utilize penumbra wedges to narrow down the scene space from which a given edge may be a silhouette. Their algorithm scales much better with sampling density than ray tracing, allowing them to render significantly faster for scenes with low geometric complexity. They use a hemicube to cache silhouette information which is highly sensitive to light orientation producing fast render times only when the light is near axis-aligned. Lehtinen et al. [41] fix many of these problems and produce impressive results even with finely tesselated geometry. To achieve these results, they introduce a BSP construction and query phase which is expensive both in time and memory relative to scene size. Both Laine et al. [40] and Lehtinen et al. [41] determine the visible depth complexity from a point, so they still must spend most of their time in a sample integration phase. My method provides higher accuracy results and raises potential performance by completely removing dependence on point sampling density.

Real-Time Methods: My work is distinct from approximate real-time techniques [30]. While these methods can work much faster than my approach or the above algorithms, they focus on plausible rather than accurate soft shadows, requiring significant approximations that break down in specific situations. Soler and Sillion [63], for example, convolve a shadow map to blur the edges of a hard shadow, resulting in some visually pleasing results. However, they have difficulties with bodies in contact and self-shadowing. Another recent body of work is precomputed radiance transfer or PRT [62, 49]. PRT methods move the visibility computations to a preprocess, and project the result into some basis (often spherical

harmonics or wavelets) for compression. With visibility already determined, the illumination can then be integrated in real-time with dynamic (usually distant) lighting environments. My method is not comparable, since it only requires precomputation of the kd-tree which has been shown to be interactive even for large scenes [36], easily allows dynamic local lighting, and can be used with general shaders. The recent work of Ren et al. [55] approximates scene geometry as a set of spheres to quickly project visibility to a spherical harmonic basis removing the need for precomputation. However, they are only able to display extremely low-frequency approximate soft shadows, whereas my system can handle exact shadows using accurate scene geometry.

2.1.2 Real-time Ray Tracing

There has been a significant amount of work focused on exploiting geometric coherence of rays to accelerate primary visibility determination. Wald et al. [76] showed that substantial benefits could be obtained by casting four rays at a time and using SSE instructions to handle these rays in parallel, reducing the number of kd-tree traversal steps and increasing memory coherence. Reshetov et al. [57] use frusta as ray proxies to accelerate kd-tree traversal. They determine the deepest kd-tree node that all of the rays in the frustum must visit, then start ray traversal there. This further reduces the number of kd-tree traversal steps, and along with extensive optimization, results in up to an order of magnitude speed improvement. Recently, Wald et al. [74] extend frustum traversal to grids.

These algorithms attempt to adapt to variations in coherence using heuristics to split the frustum along image plane axes. However, these splits are inexact for scene geometry (both acceleration structures and triangles in general orientations), and one misplaced ray is enough to slow down an entire packet. In my work, I split beams *precisely* at geometry boundaries, exploiting all available coherence with dramatic benefits for both primary and secondary effects.

2.1.3 Beam Tracing

Beam tracing was introduced in Heckbert and Hanrahan [33]. While it has found limited applicability in rendering, it has proven very useful in architectural acoustics [20, 21] where antialiasing is a primary concern. Similarly, my beam-tracing approach provides antialiasing essentially for free, and may therefore also be relevant in this domain.

Two related techniques are cone tracing [3] and ray differentials [37]. They use a partial representation of the ray's volume by including the ray's spread angle and distance to the ray's image space neighbors respectively. These elegant methods present a compromise: providing some of the benefits of beam tracing with the simplicity of ray tracing.

Most recent beam tracing methods are essentially improved or accelerated ray tracers. Ghazanfarpour and Hasenfratz [22] provide for adaptive sampling, sending beams to predict portions of an image which need higher sampling densities but fall back to rays to perform the actual visibility testing. The work of Teller and Alex [68] uses beams as a bounding volume for a single ray, then spreads the ray's results to the rest of the beam. They use splits aligned to the image axes at kd-tree and triangle boundaries to progressively improve the result.

These works and the frustum proxy methods in Section 2.1.2 circumvent beam intersection methods based on the assumption that calculating beam–geometry intersections is more expensive than tracing many extra rays. While this has been true in the past, my work removes this assumption, and enables one to explore the full power provided by precise area sampling over point sampling.

2.2 My Beam Tracing Algorithm

For simplicity of exposition, I describe the general beam tracing algorithm in the context of primary visibility. The same ideas extend to secondary shadow beams (see Figure 2.1 top and bottom).

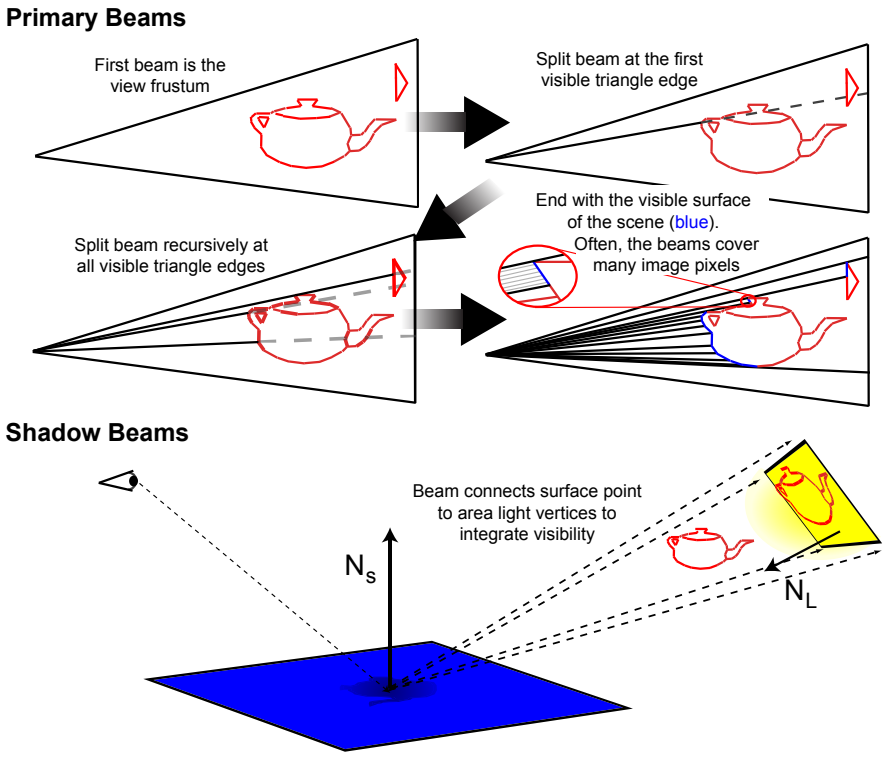


Figure 2.1: (Top) Beam tracing for primary visibility. (Bottom) Beam tracing for soft shadows.

As shown in Figure 2.1(top), I start with one large pyramidal beam representing a volume of perspective parallel rays. This beam is recursively split at geometry primitive boundaries (triangle edges) into a list of beams which is the visible surface of the scene.

As with other recent work on fast ray tracing, I use scenes that are built strictly with triangle primitives, and use a kd-tree for acceleration. Beams traverse the scene's kd-tree in a method somewhat similar to standard ray tracing. Visible triangles found in the kd-tree's leaf nodes will split the beam into two lists of sub-beams: hit beams and miss beams. The miss beams continue scene traversal until they either hit a triangle or exit the scene. As the beams split, they form a beam tree (not to be confused with that generated by reflected and refracted beams in [33]).

The primary challenges in making beam tracing efficient are (1) fast beam-triangle intersection routines, and (2) fast methods to traverse the kd-tree acceleration structure. In

this section, I give a high-level overview of my novel algorithms for these tasks. Appendix A provides more low-level details and pseudocode—interested readers will wish to follow it in parallel with this section.

2.2.1 Beam Representation

I represent my beams by 3 or 4 corner rays emanating from a common origin. Operations on these corner rays can be performed in parallel using SIMD instructions available on most modern processors (similar to Wald et al. [76]). Pseudocode for my beam representation is in Appendix A.1. It is important to note that I represent the ray directions as points on some plane as this helps with the efficient triangle intersection algorithm described below. For primary beams, I use the image plane. For secondary point light beams, I use the plane of the hit triangle, and for area light beams, I can use a plane on or near the light source. This ensures that accuracy is measured in the relevant space.

2.2.2 Beam–Triangle Intersection

Algorithm Overview: When intersecting a beam and a triangle, we seek to split the beam into two parts: that which hits, and that which misses. The beam is split by planes defined by the beam origin and the triangle edges. Most beam tracing methods do not provide details on their beam-triangle intersection tests, and generally use standard geometry set operations [33] or off-the-shelf polygon clipping algorithms. My algorithm is unique and specifically optimized for beam tracing.

High Level Decisions: My design is based on two decisions:

1. *Mirror, as closely as possible, ray–triangle intersection, diverging only where necessary.* This is aimed at keeping my algorithm as simple as possible, while benefitting from most published ray-triangle intersection optimizations.

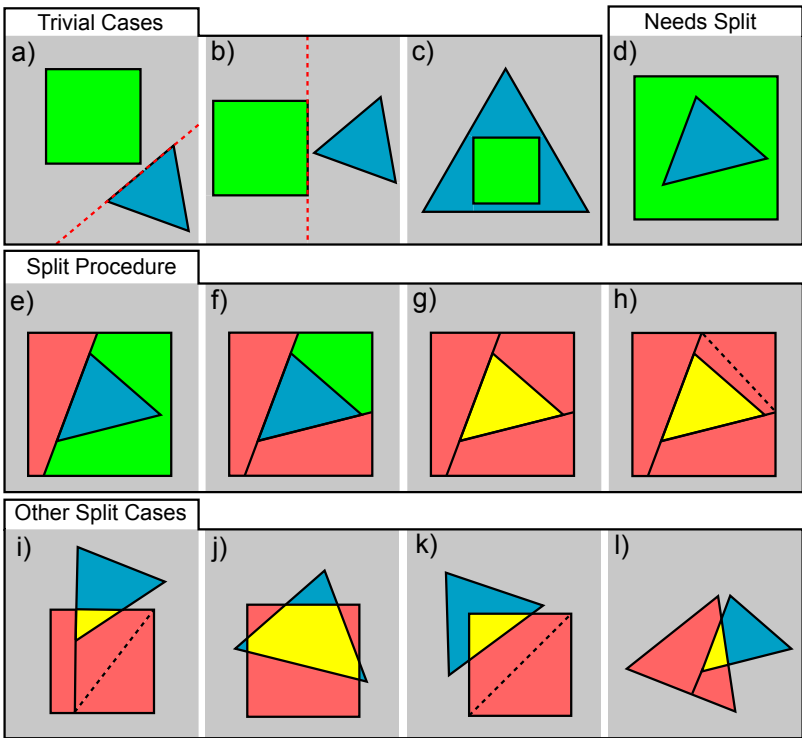


Figure 2.2: Beam–triangle intersection. The initial beam is green, the initial triangle is blue, miss beams are red, and hit beams are yellow. (a)-(c) are trivial cases (step 2). In (d), the beam must split (step 3). (e)-(g) split along each edge of the triangle. In (h), the beam needs an extra split to maintain a maximum of 4 vertices per beam. (i)-(l) show several other possible beam–triangle splits discussed in the text.

2. *Parallelize through the use of SIMD SSE instructions.* While this is not usually viewed as a high-level decision, in this case it fundamentally impacts all levels in the beam tracer, leading to an algorithm quite distinct from its serial counterpart.

Note that (2) reinforces (1) by operating on the corners of the beam as if they were a single ray. This leads to an algorithm that is as fast as ray–triangle intersection in the most common cases, and only as slow as generating $1 - 5$ new rays (in a conventional ray tracer) in the worst cases.

Step 1–Triangle Projection: As with ray tracing, I make the problem easier by solving it in 2D. However, instead of projecting the beam to the triangle's plane (the common ray approach), I project the triangle onto the image plane. This is mostly to avoid numeri-

cal problems introduced by solving each beam–triangle intersection on a different plane: intersections with neighboring triangles will not precisely agree on their shared edge's location. When I project, I must clip some triangles at a near plane parallel to the image plane, because some triangle vertices may be behind the camera's focal point. The vertices are then orthographically projected onto either the xy, xz, or yz plane depending on the image plane's normal. As noted above, the beam's corner ray directions are points on the image plane, so they need not be projected.

Step 2–Handle Trivial Cases: Figure 2.2 provides an illustration of the 2D beam–triangle intersection problem. There are 3 trivial cases as shown in Figs. 2.2a-c. In case (a), one of the triangle edges completely separates the triangle and the beam, so the beam misses the triangle. In case (b), one of the beam edges is a separating edge. Case (c) shows a beam completely contained by the triangle, since all points of the beam are inside all three edges of the triangle. Cases (a) and (c) behave the same as for rays—I use SSE to test all 3 or 4 of the beam's corner rays in parallel, such that handling these cases is similar (and as efficient) for a beam as for a single ray. Similarly, I handle the 4 beam edges in parallel for case (b).

Step 3–Beam Splitting: The rest of the cases fall into Figure 2.2d, where some of the beam is inside the triangle and some is outside. The beam now needs to split along triangle edges. I do this using a method similar to Sutherland and Hodgman [67], who clip one edge at a time. The splitting process is shown in Figs. 2.2e-h. The first edge (e) splits the beam into 2 sub-beams: one which misses the triangle, and one which partially hits and needs further processing. The second edge (f) splits the remainder into a miss sub-beam, and partial hit. After splitting along the third edge (g), I am left with a beam completely inside the triangle, and three beams which miss.

The splitting requires finding intersection points between beam edges and triangle edges. The standard Sutherland–Hodgman algorithm processes vertices one at a time. I use a parallel SSE based algorithm which handles all beam vertices at once for each triangle edge

(See Appendix A.2 for details). Computationally, finding the two intersection points for each triangle edge is equivalent to simply generating one new ray in a conventional ray tracer.

All splits are guaranteed to generate beams with convex cross-section. But as the beam splits, the cross section may become more complex, requiring more than the 4 corner rays I can handle with SSE instructions. In these cases, I perform an extra split (shown as a dashed line in Figure 2.2h), generating one 4-corner beam, and one 3-corner beam. The 3-corner beam has a triangular cross-section. I still use 4 corner rays with one being degenerate. Note that Figure 2.2d shows a relatively simple situation where the triangle lies fully inside the beam. Figures 2.2i-l show several other beam–triangle interactions. Note that all of them can be handled in the exact same way, but not all of them are split by all 3 triangle edges. Again, any extra splits added to keep the beams down to 3-4 corners are shown as dashed lines.

Robust Splitting: Robustness is a major issue for beam tracers. Precision errors in ray tracers are highly localized to the individual ray and are evidenced by black pixels and seams between triangles. Errors in beam tracers accumulate down the beam tree possibly leading to the misclassification of entire image regions. There are two enhancements I use to combat precision errors. First, I perform all intersection tests in the same plane, otherwise the same edge belonging to the different triangles may give different results even when tested against the same beam. Second, I use "fuzzy" logic tests throughout my beam tracer (See Appendix A.2 for details) to use all of the beam's corner rays to inform any decision.

Performance: The triangle intersection algorithm is relatively fast by itself. For simple scenes (those less than 2000 or so triangles) I render primary visibility in real time (10-40 FPS) without using any acceleration structure. However, as with ray tracing, beam tracing slows down very quickly without an acceleration structure. Therefore, I next introduce my beam–kd-tree traversal algorithm.

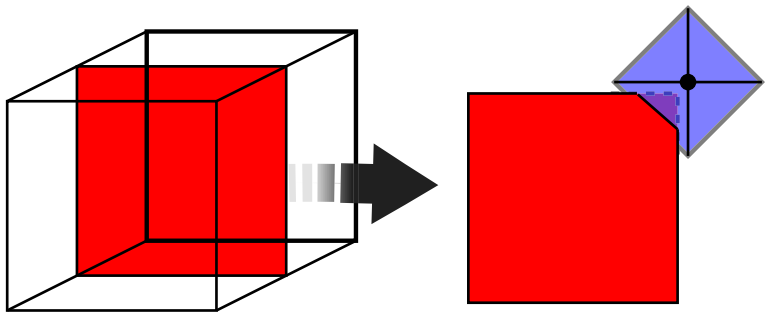


Figure 2.3: (Left) The large red plane is the current kd split plane. (Right) the split plane is isolated, the beam is in blue, and the viewpoint is down the axis of the beam pyramid. In this case, the beam's corner rays only want to traverse the far cell, but, clearly, we also need to visit the near cell.

2.2.3 KD-Tree Traversal

Ray–kd-tree Traversal: The standard ray–kd-tree traversal algorithm as introduced by Whitted [78] maintains a minimum and maximum distance along the ray. The minimum distance is where the ray enters the current kd cell's bounding box, and the maximum is where it exits. These distances multiplied by the ray's direction give intersection points with the kd cell's bounding box. At each inner kd cell, the distance to its split plane is tested. If the distance is less than the minimum distance, the ray only needs to traverse the far cell. If it is greater than the maximum distance or the plane lies behind the ray origin, the ray need only traverse the near cell. If it is in between, the ray must visit both cells.

Extending to Beams: The frustum shaped beam's near and far planes are analogous to the ray's minimum and maximum distances. However, simply using the beam's four corner rays to decide the path of all its rays can be inaccurate. This is shown in Figure 2.3 where a corner of a kd split plane pierces a face of the beam. All corner rays believe they only need traverse the far cell, when clearly some of the rays in the pierced face also need to visit the near cell (Reshetov et al. [57] also describe this situation).

Figures 2.4a-c show my solution to this problem in 2D. The "active" portion of the beam (the portion which is traversing the current kd cell) is dark gray. The near plane is **tmin** (red)

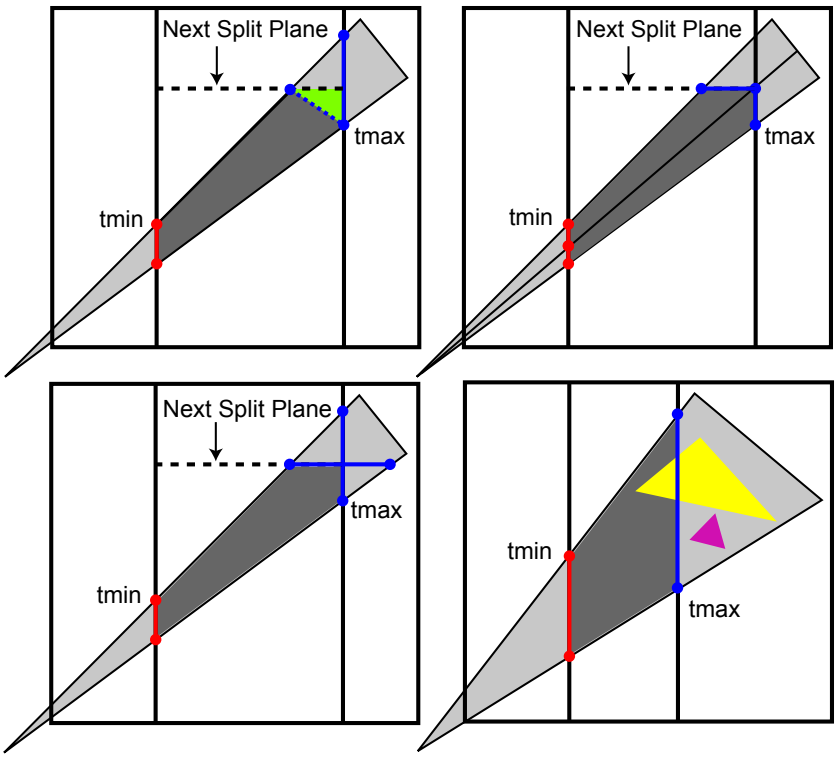


Figure 2.4: (a) The minimum and maximum distances for all of a beam's rays cannot be represented by single near and far planes. If we just used the corner rays' far distances, we wouldn't be including the green portion of the beam. (b)We could split the beam with a new far plane. (c) Or we can use multiple (3 for 3D) near and far planes. (d) The large yellow triangle will be intersected before the small purple triangle even though the purple is closer. The hit beam for the yellow triangle must continue traversal.

and far plane **tmax** (blue). The next kd split plane is shown as a dashed line. If we attempted to use just the corner rays' maxima, we don't have enough information to represent the green portion of the beam. While this doesn't cause a problem in 2D, it leads to situations like the one in Figure 2.3 for 3D. One possible solution would be to split the beam where the kd planes split the far plane as in Figure 2.4b. This approach is precise in that it traverses all of and only the cells which intersect the beam. I tried this approach with positive results, but there is significant overhead in the beam splitting operations so it does not scale well with larger kd-trees, and dramatically reduces available coherence.

My Solution: Instead, I use the much simpler approach in Figure 2.4c inspired by frustum proxy methods like MLRT [57] and LCTS [32]. I maintain 3 near and 3 far planes, one

for each axis. This is like keeping track of 3 beams, where the "active" ray volume is their intersection. If the beam's corner rays' distances to the current split plane are all less than the distances to either of the other axes' near planes, the beam traverses only the far node. Likewise, if the distances are all further than either of the other axes' far planes or lie behind the beam's origin, the beam traverses only the near node. Otherwise, the beam must visit both nodes. This leads to only two extra comparisons in the traversal code (see pseudocode in Appendix A.3).

Handling Hit Beams: My algorithm for walking a beam through a kd-tree is not much different from that for a ray, with some notable exceptions. The biggest difference, shown in Figure 2.4d, is how hits are handled. Once a ray finds a hit in a leaf node that is closer than its maximum exit distance from the leaf node's bounding box, that hit is guaranteed to be closer along the ray than any triangle in any other leaf node. A beam hit, on the other hand, may include a triangle that is not wholly contained by the current kd cell (the yellow triangle in Figure 2.4d). There may be another triangle (the purple triangle) inside the beam which occludes the hit triangle but resides in a further leaf cell. While we could clip all hit beams to the planes of the kd cell to assure that we get only the parts of the triangles that are guaranteed to be closest, this leads to excessive fragmentation of the beam. Instead I keep the full hit beam which must continue kd-tree traversal until it reaches a kd cell which is wholly further than the hit beam.

Comparison to Frustum Proxies: Once the beam is split, the sub-beams continue scene traversal from the *leaf node* where they are generated. This is an important distinction from frustum proxies for ray tracing, such as Reshetov et al. [57]. Those methods merely find a deep sub-tree within the kd-tree from which to start shooting rays. Each ray or ray packet must start from the root of this sub-tree leading to more redundant kd steps. Moreover, the frusta must often visit even more nodes to find a suitably *deep* one.

It is possible that one of the sub-beams may start from a leaf node that it would not

have visited if it had not been led there by its parent. However, in practice, this is quite rare because I split *exactly* at geometry boundaries. This substantially reduces redundant kd steps, increases memory coherence, and thus minimizes my performance dependence on absolute scene size.

2.2.4 Miscellaneous Acceleration Techniques

Since we expect to visit far fewer kd leaf nodes than for ray tracing (or at least visit them much less frequently), we can expend a little more time there to reap some benefits.

Leaf Cell Optimizations: Upon reaching a leaf node, I do one processing pass on the triangles before performing any intersection tests. It is during this pass that I project the triangles and split ones that don't fully project to the image plane. Since each triangle vertex only needs to be projected once per frame, I maintain a list of projected triangle vertices. I mark triangles that are either backfacing or fully behind the beam's near plane so that I needn't bother accessing them with later beams in the same frame. I also found that sorting the triangles in the leaf nodes according to surface area, with larger triangles first, leads to less splitting and better performance. I perform this sorting during the kd-tree build.

Mailboxing For Beams: Wald et al. [74] observe that while mailboxing often adds more overhead than benefit for a ray tracer, it becomes increasingly valuable for ray bundles and frusta. However, mailboxing is also more complicated for my beams since a triangle may have been previously tested against the current beam or any of its parents. To deal with this, I maintain a hierarchical mailboxing structure (I call it the Post Office) where each node is just an integer pointing to its parent node. Each level of the Post Office represents the new beams generated at a kd-tree leaf. Each beam contains its own mailbox ID, and each triangle is marked with the current beam's ID. The mailbox test checks whether the triangle's mailbox ID matches either the current beam's ID or any of its parents' IDs. It turns

out that it is excessive to test every parent ID since it is much more likely that a triangle was intersected with one of the 3 or 4 immediate parents of the current beam. I test against the current beam and its 3 parents. Mailboxing gives us a speed improvement between 10% and 50%.

2.3 Beam Tracing Analysis–Primary Rays

I now study the performance of beam tracing for real-time primary visibility and point lighting, which is very useful in understanding its overall performance characteristics. Those readers more interested in my application to soft shadows are referred to Section 2.4.

For my beam tracer, I trace primary visibility, then send the hit beams to the GPU as a list of quads for final rasterization. Since I send full area elements, I can render with 6× antialiasing and shade with a general per-pixel programmable shader. These area elements exactly represent the visible surface of the scene, most often with many fewer elements than the total number of triangles in the scene. As such, even at 1024x1024 resolution with antialiasing and per-pixel shading, the GPU portion of my algorithm uses almost negligible time. By contrast, optimized ray-tracing techniques return point samples and can show as much as 10%-33% overhead just for cosine shading [57]. To focus my comparisons on visibility computation rather than shading, in this section I use only (antialiased) diffuse shading and one light positioned at the camera.

I compare Beams to both my own optimized ray tracer (Rays, Section 2.3.1) as well as MLRT [57] (Section 2.3.2), the fastest current method. I use MLRT's kd-tree builder (with settings tuned specifically for MLRT's performance) for beam tracer, ray tracer, and MLRT alike. I don't use antialiasing for either my ray tracer or MLRT as this would slow down the performance of those systems. All images are generated at 1024×1024 resolution on a 3.0 GHz Pentium 4 processor with 1.5 GB of memory and an ATI Radeon 9800 graphics card.

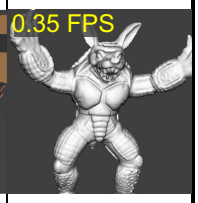
Scene / # Triangles / Thumbnail / Average Measurements		Erw6 (816)	Soda Hall (2,195K)	Conference (282K)	Armadillo (345K)
					
Visible Tris./Frame	Beams	150	1,536	4,602	25,647
KD Steps/Pixel	Beams	.0085	.13	.30	1.07
	Rays	13.86	42.36	25.67	46.46
Intersections/Pixel	Beams	.0033	.064	.48	.6
	Rays	6.81	4.29	16.34	5.67
Hits/Pixel	Beams	.00096	.0097	.027	.14
	Rays	.95	.98	.91	.99
Frames/Sec.	Beams	**169.49**	**21.28**	**5.56**	**1.72**
	Rays	**.99**	**.52**	**.52**	**.44**
	MLRT	**12.99**	**5.56**	**7.14**	**5.99**
Visible Tris./Sec.	Beams	25,424	32,686	25,587	44,113

Figure 2.5: Comparison of statistics for beams and my optimized ray tracer. The numerical values are averages over a large number of views, all rendered with diffuse shading and a single point light at the camera. (The fps numbers on the thumbnails on top correspond to that particular view.)

2.3.1 Beams vs. Rays

Comparison Setup: I compare to my own optimized one-at-a-time ray tracer which uses cache coherent data structures and optimized kd-tree traversal and triangle intersection. While it doesn't use ray bundles and frustum proxies, it is competitive with other fast one-at-a-time ray tracers—I will verify this by comparing timings with MLRT in Section 2.3.3. I also use this ray tracer later for my comparisons with secondary soft shadow rays in Section 2.4.

The table in Figure 2.5 shows several statistics for both beams and rays on several scenes. The values in the table represent averages taken over many viewpoints in the scene. The number of kd-tree steps and intersection tests are the total for rendering a single frame, divided by the image resolution (1024x1024).

Performance Comparison: For the simplest scene (Erw6), my beam tracer performs several orders of magnitude fewer kd steps and intersection tests leading to framerates well into the hundreds. More impressively, beams achieve high framerates on the soda hall model with over 2 million triangles. While my average framerate is about 21 FPS, it often stays in the hundreds while on the inside.

Even for the highly tesselated conference model, beams perform one to two orders of magnitude fewer kd steps and intersection tests, maintaining real-time performance. The heavily tesselated Armadillo model is a worst case scenario for my beam tracer with many small triangles which are often smaller than a pixel. Even so, this model shows that my beam tracer is robust enough to handle even highly complex models.

Analysis: There is some worry that the beam splitting process could lead to exponentially more beams than the number of visible triangles, leading to an exponential decay in performance as we move to larger scenes. In Figure 2.6, I plot visible triangles versus hit beams to show that this is not the case. The ratio of hit beams to visible triangles stays fairly constant at around 5.5–6.5.

Figure 2.7 plots visible triangles per frame versus wall-clock time. Performance is almost perfectly linear in the number of visible triangles (or hit beams), but is relatively insensitive to absolute scene size. (The slope of the conference model graph is steeper showing some connection to scene depth complexity.) Indeed, my beam tracer is faster on the 2 million triangle soda hall model than on the 280K conference model. For soda hall, only the first beam needs to walk all the way to the leaves of the kd-tree and the rest spend most of their time at the leaves, which is where we want them to be.

2.3.2 Beams vs. MLRT

Comparison Setup: I now compare timings against MLRT [57], which is currently the fastest known ray tracer. MLRT uses a frustum proxy mechanism to find deep kd-tree entry

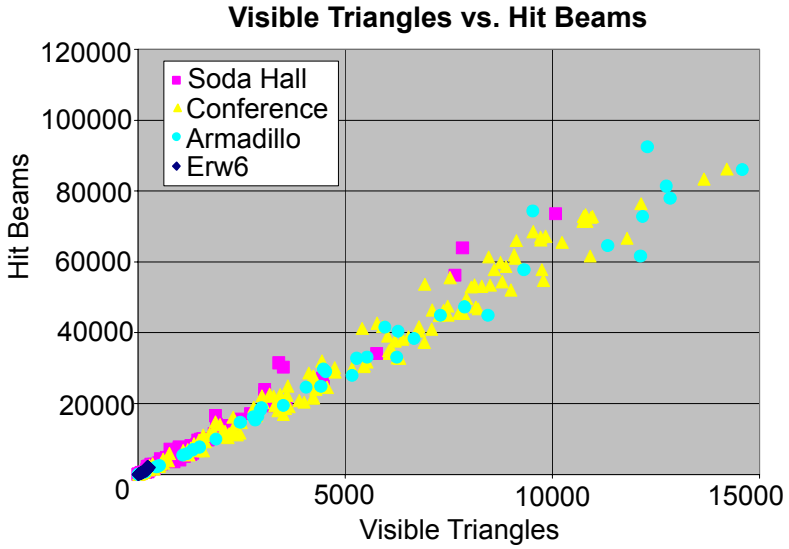


Figure 2.6: Number of visible triangles vs. number of hit beams, showing a simple linear relationship. The data points correspond to different views.

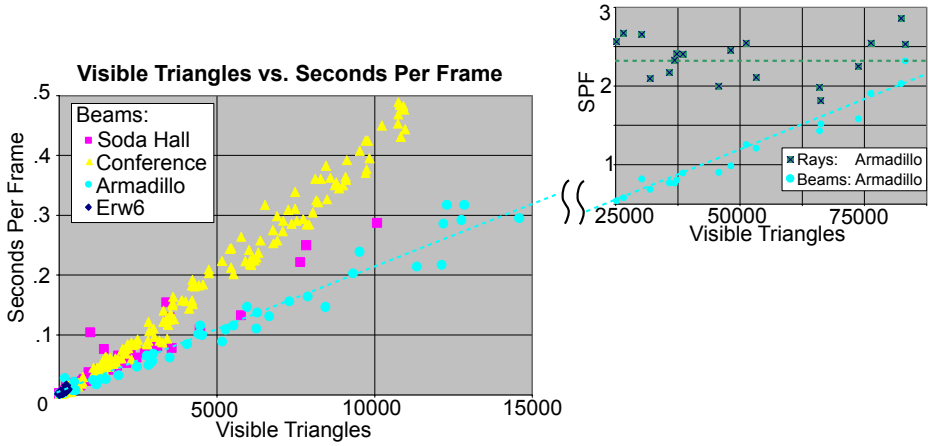


Figure 2.7: Number of visible triangles vs. seconds per frame. Running time is proportional to the number of visible triangles for my beam tracer. Different data points correspond to different views of the scenes.

points from which it sends 4x4 packets of rays.

To get closer to a direct algorithmic comparison, I set MLRT to use only one thread and turn off quad generation (MLRT aggressively groups axis-aligned triangles into large quads allowing the use of a greatly simplified intersection test). I believe beam tracing would benefit equally from these two optimizations, and that the complexity in implementing them

would be comparable. Image tiling for parallelization is as easy for beams as rays, and MLRT needs to implement the quad optimizations for multiple ray types (frusta, 4x4 packets, 4x1 packets, etc.) whereas I deal only with beams. I summarize how these optimizations affect MLRT's performance in Figure 2.8.

Timing comparisons are shown at the bottom of Figure 2.5. Note that these are averages over many views as usual. (They are similar but not exactly the same as in Figure 2.8, since the latter were taken from rendering a single view, to enable easier comparison to published MLRT results.) I wasn't able to measure the number of kd steps and intersection tests for MLRT, but it is possible to estimate these numbers by taking my ray tracer's measurements (Figure 2.5), and adjusting them downward to account for the known improvement MLRT provides. Specifically, to estimate MLRT's kd steps, divide my ray tracer's measurements in Figure 2.5 by about 10, and for the intersection tests, divide by about 4. This gives a fairly accurate estimate, agreeing with MLRT's published results. Since MLRT traces four rays at a time, the number of intersection tests and kd steps should be divided by 3 or 4. Beyond this, the kd steps should be further divided by 2.5 - 3.5 to account for MLRT's frustum traversal. These adjustment factors are taken from Reshetov et al. [57].

Performance Comparison: From Figure 2.5, I am over an order of magnitude faster than MLRT for Erw6. My most impressive result is on soda hall, where I am 4× faster. The absolute size of the model slows MLRT down in the kd-tree's inner nodes, while my beams keep to the leaves. While speed is comparable to MLRT for the highly tesselated conference model, I still perform an order of magnitude fewer kd-steps and intersection tests. I perform fewer such tests even on the armadillo where the beam tracer is slower.

Point Light Hard Shadows: The true power of the ray tracing body of algorithms is high quality secondary effects such as shadows. However, gathering the primary pixels into large enough coherent groups for either ray packets or frustum proxies to be efficient is a difficult task. For my beam tracer, I simply connect my primary hit beams, which often contain

# Threads	Quad Optimization	Shading + Rendering	Erw6	Soda Hall	Conference Room	Armadillo
2	On	Off	87.58 FPS	20.42 FPS	15.23 FPS	10.34 FPS
1	On	Off	74.12 FPS	16.26 FPS	10.85 FPS	6.98 FPS
1	Off	Off	27.35 FPS	8.97 FPS	7.22 FPS	5.23 FPS
1	Off	On	13.05 FPS	7.36 FPS	5.6 FPS	4.35 FPS

Figure 2.8: Impact of multi-threading, quad optimizations, shading and rendering on MLRT. The timings were taken from rendering a single view. To make a fair algorithmic comparison for rendering and displaying diffuse shaded images, I consider the last line for my tests.

10s to 1000s of pixels, to the point source with another beam. When I trace this beam, the resulting hit list represents obstructed portions of the beam, while the miss list is fully unobstructed.

The images in Figure 2.9 were rendered with my beam tracer at 1024x1024 inside the soda hall model at 21 FPS. MLRT renders this view at only 5 FPS. The left image shows the beam tracer's high quality shadows, while the right image shows the beams which created it—notice that the beam tracer produces a perfect shadow cut-out for high quality antialiasing. I display a close-up of the yellow highlighted region for both beam tracing and MLRT in Figure 2.10. The MLRT image (right) clearly shows jagged object and shadow edges along with spurious black pixels. The beam tracer's image is rendered with 6× antialiasing with almost no performance penalty.

2.3.3 Rays vs. MLRT

Before moving on to soft shadows, I take a moment to compare MLRT to my own ray tracer, since I use the latter for comparisons in Section 2.4. From the bottom of Figure 2.5, MLRT is consistently about 10×–14× faster. This is as expected since both ray packets and frustum traversal introduce a multiplicative factor of about 3–3.5 each, and MLRT is also strongly optimized for memory efficiency.

I therefore conclude that my ray-tracer is quite well optimized for a one-at-a-time tracer.

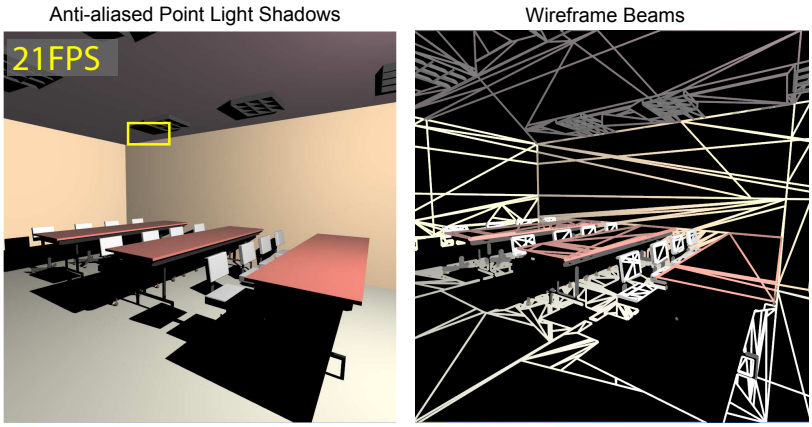


Figure 2.9: Real-time antialiased point light shadows. (Left) This image was rendered at 21 frames per second at 1024x1024 resolution with 6× antialiasing inside the 2 million triangle soda hall model. (Right) The wireframe beams which generated the top-left image: all geometry and shadows are cut out exactly providing for high-quality antialiasing (Figure 2.10).

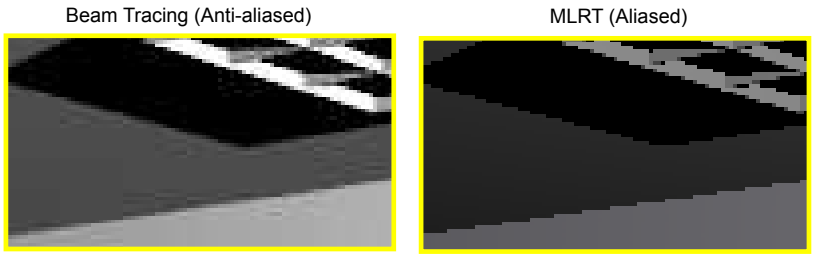


Figure 2.10: (Left) close-up of highlighted region from Figure 2.9. I render with 6× antialiasing with almost no performance penalty. (Right) same region with MLRT clearly displaying jagged edges and spurious black pixels.

Moreover, MLRT's speedups are specific to primary visibility. It is unclear as to how these methods perform for secondary effects like area lighting, and it is expected that the speedup will be much smaller (a brief discussion of the arguments for this are given in Section 2.5.1). Therefore, I use my heavily optimized ray tracer as the comparison method in Section 2.5.

2.4 Soft Shadows

I now describe the main application of my beam tracer, to efficiently compute accurate soft shadows. To compute soft shadows from an area source, we need to solve the area lighting

equation

$$B(\mathbf{x}, \omega_o) = \int_A V(\mathbf{x}, \mathbf{p}) L(\omega_i) \rho(\omega_i, \omega_o) \cos\theta_i \cos\theta_l \, d\mathbf{p}, \tag{2.1}$$

which gives the exitant radiance at a point $\mathbf{x}$. $V(\mathbf{x}, \mathbf{p})$ is the visibility from $\mathbf{x}$ to a point on the light $\mathbf{p}$, and $\rho(\omega_i, \omega_o)$ is the BRDF. θ_l is the angle made by the incoming ray with the surface normal at the light, and we integrate over the area A of the light source.

The most difficult part of solving this equation is the determination of the visibility $V(\mathbf{x}, \mathbf{p})$, between each image point and all points on the light source, and this is where beam tracing can be most useful. For a simple triangle or square light source, I simply create a beam whose apex is the image point and base is a triangle or quad of the light source (See Figure 2.1 (bottom)).

After shooting the beam, the beam trace's hit list represents obstructed polygons on the light source while the miss list is visible polygons on the light source. There are two options for calculating the irradiance from the visible portion of the light. We can add up the contribution from each of the polygons in the miss list, or first calculate the lighting as if the entire light were visible and subtract the contribution of the polygons in the hit list. Both methods are equivalent, and I choose one based on which list is smaller.

For simplicity of comparisons in Section 2.5, I integrate the lighting using the common approximation also used by Soler and Sillion [63] and Agrawala et al. [1]. I modulate the irradiance of a point source at the center of the light with the fractional visibility of the entire area light. This allows us to pull the lighting and BRDF terms as constants outside the integral and focus on visibility,

$$B(\mathbf{x}, \omega_o) = L\rho(\omega_i, \omega_o) \cos\theta_i \cos\theta_l \int_A V(\mathbf{x}, \mathbf{p}) d\mathbf{p}, \tag{2.2}$$

Note that this still requires integrating over the entire visibility function to find the average attenuation of the area light source.

It is also possible to integrate the lighting exactly using either Hart et al.'s method [29]

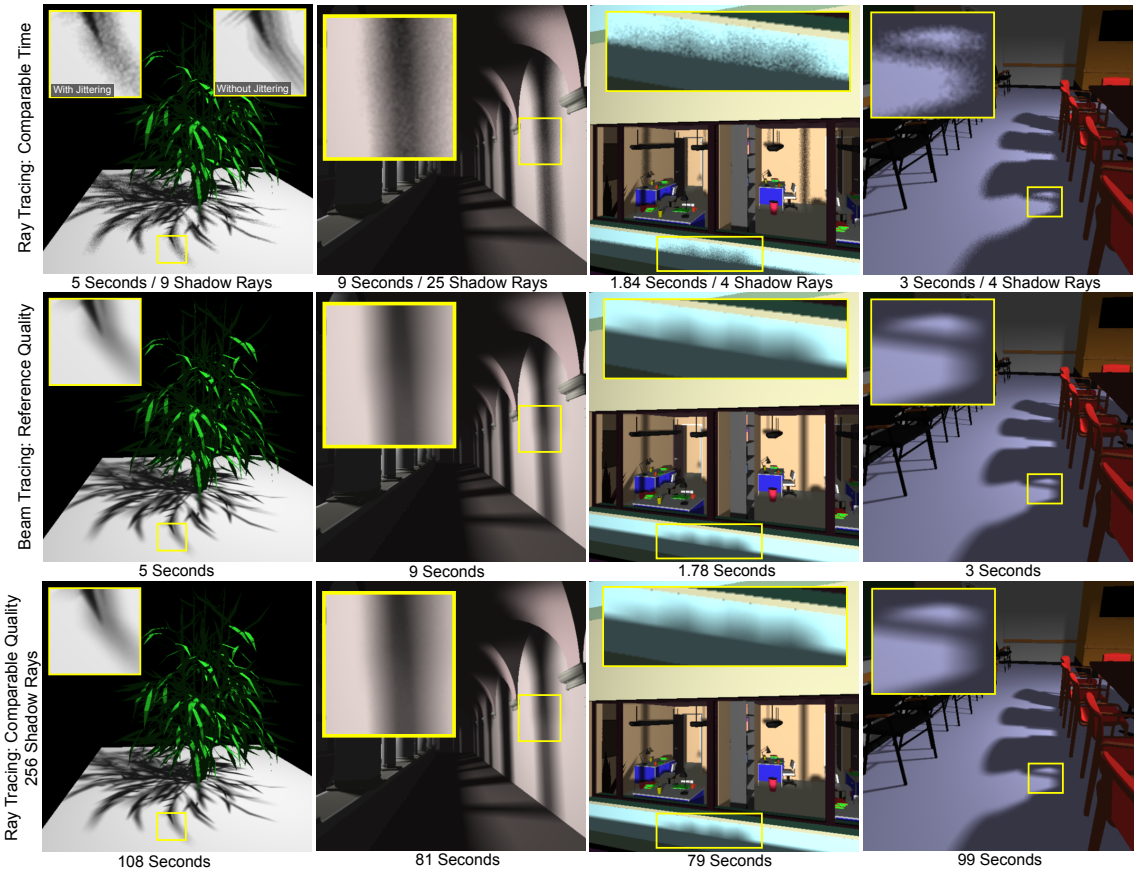


Figure 2.11: Beam tracing vs. ray tracing. Beam tracing (center row) provides an exact lighting solution in seconds. Ray tracing requires 256 samples or shadow rays (bottom) to reduce noise within tolerance. Attempting to ray trace with few enough samples to match the speed of beam tracing (top) leads to severe noise. All images are rendered at a resolution of 512×512 pixels.

for Lambertian surfaces or Arvo's [4] for specular. However, I found that equation 2.2 works well for small area lights with diffuse objects. It also reduces noise in the ray tracing comparison method by removing the samples' shading dependence on light location.

2.5 Results

Comparison Setup: To render an area lit image, I first need to trace primary visibility for which I use my ray tracer. All timings include time to cast primary rays, as well as shadow casting and calculation. All tests were run on a PC with a 3.0 GHz Pentium 4 processor, 1.5

GB RAM, and an image resolution of 512x512 (for soft shadows, timings for both beams and rays scale linearly with image resolution, and their ratio remains relatively unchanged).

My beam tracer consistently produces noise-free results regardless of scene complexity and sample density so it is difficult to directly compare to ray traced solutions. I do so here to provide a context to measure performance. As such, determining the correct number of samples and sampling strategy for "comparable quality" results is somewhat arbitrary. I use 256 samples on a jittered grid as this is often considered the minimum requirement to produce reasonable quality soft shadows.

I compare to my own optimized ray tracer. As demonstrated in Section 2.3.3, it is well optimized for a one-at-a-time ray tracer. I do not use MLRT because it does not provide a suitable area lighting implementation. Besides, as described in Section 2.5.1, it is unclear whether it would provide significant benefits. Since I seek to study the effectiveness of secondary visibility determination, I use Lambertian materials and a single square light in all comparisons. I use my own kd-tree builder (with the same tree for beams and rays) for all scenes except soda hall. It is constructed using the most basic form of the surface area heuristic construction algorithm as described in Havran's thesis [31], and the nodes use a cache optimized layout as in Wald et al. [76].

Scenes: The plant scene has 5245 triangle faces. Almost every edge is a silhouette edge making it difficult for edge based methods such as Laine et al. [40] and Lehtinen et al. [41]. The soda hall model, with well over 2 million triangles, would require occlusion culling in order for these methods to handle this scene efficiently. The Sponza Atrium (76154 triangles) and conference (282801 triangles) represent mid to large sized scenes. The light source size and camera viewpoint were selected to show interesting configurations for generating soft shadows.

Image Comparison: Figure 2.11 compares image quality and wall clock time using my beam tracer vs. ray tracing. The center row, generated using the beam tracer, serves as the

Measurements \ Scene # Triangles		Plant (5,245)	Sponza (76K)	Conference (282K)	Soda Hall (2,195K)
Visible Tris./Pixel	Beams	.90	7.32	2.07	.81
KD Steps/Pixel	Beams	19.38	68.90	26.59	46.34
	Rays	1,766	4,270	4,289	7,848
Intersections/Pixel	Beams	65.01	100.42	19.62	6.09
	Rays	5,741	2,641	4,133	242.13
Hits/Pixel	Beams	1.67	19.00	3.64	1.59
	Rays	24.59	92.22	72.36	48.07
Secs./Frame	Beams	**5**	**9**	**3**	**1.78**
	256 Rays	**108**	**81**	**99**	**79**
Visible Tris./Sec.	Beams	47,186	213,211	180,879	119,290

Figure 2.12: Beam tracing performance for soft shadows. I show several statistics for generating the images in Figure 2.11. I compare the performance of my beam tracer with my ray tracer (the latter with 256 shadow rays). The beam tracer uses significantly less kd steps and intersection tests leading to at least an order of magnitude improvement.

reference result since I always obtain an exact solution for the visibility. A minimum of 256 shadow rays, bottom row, are required to reduce noise to acceptable levels. As can be seen in all examples in the top row, reducing the sample count leads to severe noise. For the plant image, I also include a close-up without jittering, to show alternative banding artifacts—the noise from jittering is generally considered less disturbing than banding. In these examples, the beam tracer achieves a 10×–40× improvement over ray tracing for "comparable" image quality.

Quantitative Comparison: Figure 2.12 shows several statistics for beam tracing and ray tracing (with 256 shadow rays). I divide all statistics by the image resolution (512x512) to give per pixel measurements. I also include the number of visible triangles seen from each pixel by my beam tracer for the shadow beams, since the number of visible triangles was identified as a primary scene component relating to my performance in Section 2.3.

I am clearly operating well into the region where beam tracing offers its greatest impact. With an average of less than ten visible triangles from each pixel for these scenes, my beam tracer can operate on hundreds of thousands of image pixels per second with high accuracy.

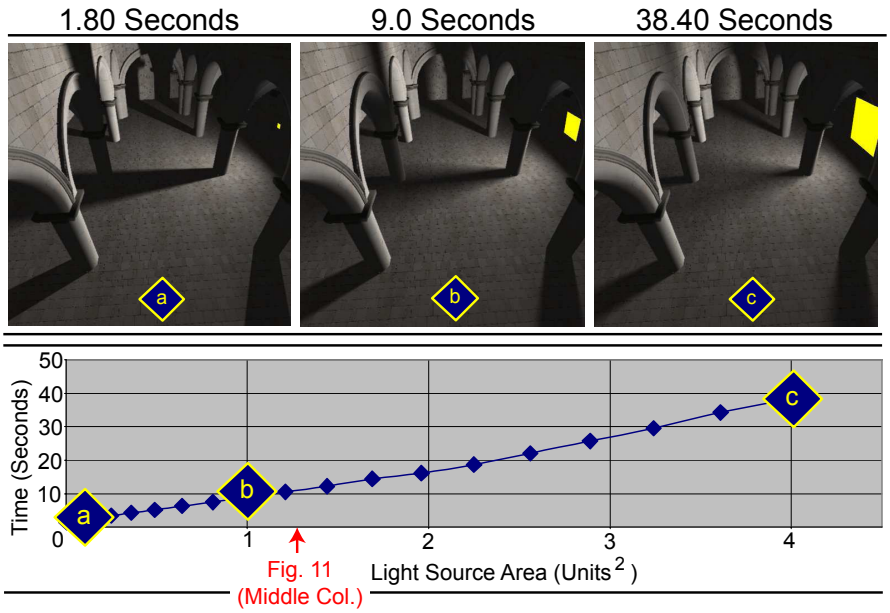


Figure 2.13: Beam tracing scalability with light source size. The beam tracer's results are nearly linear in the light source size.

Also note that secondary beams are able to process many more triangles per second than primary beams (compare visible triangles per second in Figs. 2.5 and 2.12). Recall that for primary visibility, the number of hit beams was about 6× the number of visible triangles (Figure 2.6). This is no longer the case for secondary visibility. It is around 2 and often less. This is because we no longer care about the nearest hit, but rather any hit between the primary hit point and the light source. Since the triangles are sorted by size within the leaves, I can directly conclude much larger light source areas as occluded. This constant is a key for measuring beam splitting efficiency, and it is clear that beams are particularly efficient for secondary visibility. So, while rays benefit from fewer rays and a first hit criterion, I benefit even more by being able to quickly eliminate more of the light source.

Light Source Area vs. Time: Figure 2.13 shows light source area versus wall clock performance using my beam tracing method in the sponza scene. Even the smallest light source in the figure generates a visible penumbra region, requiring many rays to trace accurately in a ray tracer. My results are nearly linear in the light source size. (It is difficult

to compare this graph to the behavior of other shadow algorithms with light source size, since most previous work does not report this vital statistic.)

Note that my ray tracer renders these images in around 94-98 seconds with 256 shadow rays, so even for a large light source, the beam tracer is more than 2× faster. Note also that as the light source size (and penumbra region) grows, more shadow rays than 256 would likely be needed to achieve the same level of accuracy with jittering, or reduce banding artifacts without jittering. On the other hand, the beam tracer always produces exact results, never needing to point sample the light source.

2.5.1 Comparison to Other Methods and Limitations

MLRT: Although results on MLRT applied to area lighting haven't been published, it is easy to envision a simple extension of it. While I showed that MLRT can be 10×–14× faster than my ray tracer for primary visibility, I don't believe it would be as efficient for soft shadows. When calculating primary visibility, I am dealing with hundreds of thousands to millions of samples where MLRT's frustum proxies can accelerate much larger (32x32 – 128x128) groups of rays deep into the kd-tree. With only 16x16 rays, it is hard to imagine getting much more than the 3×–4× provided by tracing 4-ray packets, not the 10×–40× improvement that I demonstrate.

Limitations: My method takes advantage of all geometric coherence available leading to fast render times when the triangles are large relative to the sample resolution. Unfortunately, this can also become a problem when there is little geometric coherence. If the scene is highly tesselated, i.e. if the visible triangles to sample density ratio is high enough (such as scenes with millions of visible triangles), standard Monte-Carlo ray tracing or the soft shadow volume method in Lehtinen et al. [41] may provide faster results, but cannot guarantee the same quality since I produce exact shadows without noise. Also, sampling based methods can handle more general situations (textured light sources, complex BRDFs,

...).

In practice, level of detail could be used (and my approach gives it even greater importance) to avoid scenes where triangles are small relative to the required sample density. This presents an application specific choice: to ray trace with reduced sampling density or beam trace with reduced geometric detail. In the future, I plan to evaluate LOD methods, and some approaches for falling back on standard ray tracing when the beam's cross-section gets too small or for evaluating more complex materials and/or lights.

2.6 Conclusions and Future Work

I have introduced a new beam tracing algorithm, making this historically slow method competitive with the fastest ray tracers for determining primary visibility for scenes with moderate complexity. Using this beam tracer, I compute exact and noise-free soft shadows in a matter of seconds.

I have only begun to optimize my beam tracing implementation. The results in Figure 2.5 give us hope that beam tracing will eventually be faster than the best ray tracing methods for all scenes except those for which the triangle size is close to sampling density.

Instead of only exploiting angular coherence from each pixel for soft shadow generation, I would also like to exploit image space coherence. I have already started exploring connecting the area light source and the image space area elements by specific beams, to exactly determine visibility obstructions between the two. The result is a discontinuity mesh relating the light source and the visible triangles. This intuitively extends to global illumination.

Beam tracing has the potential to efficiently create a variety of high quality visual effects, beyond soft shadows. Generating specular reflections and caustics is one promising direction. In fact, the work of Liu et al. [43], developed concurrently with mine, extends beams to nonlinear reflection and refraction transformations. Caustics often require millions of rays

to get a sampling density high enough to remove noise from the image. Just as for soft shadows, beams may be relatively insensitive to this problem.

Beam tracing has largely been ignored by the rendering community recently due to its perceived poor performance characteristics. My work proves that this perception is largely undeserved, and I have only provided a peek into the true potential for high speed beam tracing. I hope this work encourages more attention in this promising direction.

Chapter 3

Large Ray Packets for Real-time Whitted Ray Tracing

While a beam tracer is a powerful tool for many scenes, a ray tracer is still preferred for more general rendering problems. As shown in Chapter 2, beam tracing may be slow when there are very many small geometric primitives. Moreover, certain light transport effects – such as refractions and reflections from curved surfaces – generate non-linear ray volumes, which are difficult for a beam tracer. In these cases, it may be necessary fall back to a ray tracer.

In order to take advantage of coherence with rays instead of beams, current research suggests to use bundles of coherent rays, called ray packets. Recent approaches, such as Reshetov et al. [57], Wald et al. 2006 [74], and Wald et al. 2007 [72], allow algorithmic amortization across large packets of 16–256 rays by using new algorithms for scene traversal and bounding frusta to cull away expensive per-ray operations.

In this chapter, I aim to employ large ray packet algorithms to achieve real-time Whitted ray tracing, but ray coherence is much less reliable in this domain. Beyond primary visibility, Whitted ray tracing requires secondary rays for point-light shadows, reflections, and refractions. According to Mansson et al. [45], ray coherence degrades for these secondary

	Masked Traversal	Ranged Traversal	Partition Traversal	FrustumCulling
Introduced By	[76]	[72]	***New***	Primary+Shadows: [72] Reflections+Refractions: ***New***
Ray Packet Size	Only good for up to 4× 4 packets.	Usually best for up to 8×8 packets and sometimes 16×16.	Good for all packet sizes.	Benefits tend to increase with packet size.
Scene Complexity	Only good for simple scenes.	Best for simple to moderate scenes.	Best for complex scenes.	Benefits decrease with scene complexity.
Ray Recursion Complexity	Bad for secondary effects.	Best for primary and shadow rays. Okay for low recursion. Bad for deep recursion.	Best for deep reflections and refractions.	Up to 2× performance benefit for primary and shadow rays and 1.2×–1.3× for reflections and refractions.
Summary	Superseded by Ranged and Partition traversal.	Best for packet sizes $\leq 8 \times 8$, simple to moderate scenes, and moderate ray recursion complexity.	Best for large packet sizes $\geq 16 \times 16$, complex scenes, and high recursion complexity.	Best for primary rays and shadow rays. Helpful for reflection and refraction rays.

Table 3.1: Conclusions from the study of large ray packet algorithms in Section 3.4.

effects, and I expect a corresponding drop in performance for large ray packet algorithms.

I study the two fundamental approaches for accelerating large ray packets: ray packet acceleration structure traversal algorithms and frustum culling. In Section 3.2, I will describe two old algorithms for traversing a Bounding Volume Hierarchy (BVH) as well as one new one. I call the two existing algorithms Masked traversal (introduced by Wald et al. 2001 [76] and used in Reshetov et al. [57]) and Ranged traversal (introduced by Wald et al. 2007 [72]). I introduce a new algorithm called Partition traversal which is robust to degradation in ray coherence. In Section 3.3, I review methods to generate frustum bounds around primary and shadow rays and introduce a new method to create frustum bounds around reflection and refraction rays. I believe this is the first published work to demonstrate frustum culling for reflections and refractions in a ray packet tracer.

In Section 3.4, I examine how the large ray packet algorithms for acceleration structure traversal and frustum culling respond to changing the three primary variables which affect ray casting performance: ray packet size, scene complexity, and ray recursion complexity. Larger ray packets provide more opportunity for algorithmic amortization, but can also lead to greater divergence during scene traversal. Scene complexity involves a combination of the gross number of scene triangles, relative triangle sizes and distribution, and surface variations. Ray recursion complexity is the degradation in ray coherence caused by secondary rays and

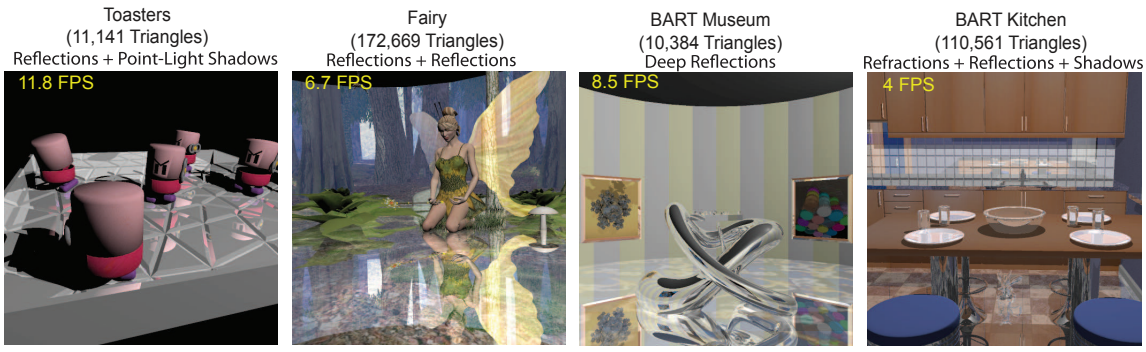


Figure 3.1: Real-time Whitted ray tracing on a single, affordable workstation is now possible. All images were rendered at 1024×1024 on a dual quad-core system (8 cores total) with 2.0 GHz Intel Xeon processors.

increases from primary rays to shadow rays to refraction rays to reflection rays and also increases with the depth of reflection and refraction recursion. I summarize my conclusions in Table 3.1.

Based on the results of Section 3.4, Section 3.5 combines the best algorithms to create my fully real-time Whitted ray tracing system which consistently provides performance benefits of 3×–6× over 2×2 SIMD ray packets (see Table 3.2). With affordable multi-core CPU technology multiplying almost another order of magnitude, real-time Whitted ray tracing on commodity hardware using a single workstation is now fully realized. The images in Figure 3.1 were all generated with my system. From left to right, the first image has a reflective floor and point-light shadows and runs at 11.8 FPS. The fairy in the second image has refractive wings. Her eyes and the gold and jeweled components on her wand as well as the forest floor all reflect the scene. Even on this complex scene, I achieve 6.7 FPS. The frame from the BART museum in the third image is an example of deep reflections using 3 bounces of reflection at 8.5 FPS. Lastly, the frame from the kitchen scene at the far right puts it all together, using 1-bounce reflections, 4-deep refractions, and point-light shadows at an interactive rate of 4 FPS.

3.1 Background

I introduce the algorithms I will be studying in Sections 3.2 and 3.3. First, I review previous work and provide some background for understanding large ray packet algorithms.

3.1.1 SIMD Ray Packets

SIMD ray packets were first used by Wald et al. 2001 [76] and allow 4 rays to traverse the scene as if they were one by taking advantage of the Single Instruction Multiple Data units available on modern CPUs. 4-wide SIMD ray packets consistently provide a 2×–3× performance improvement over single rays.

In my system, I use 2×2 SIMD ray packets as my smallest ray primitive and will refer to a 2×2 ray packet as an individual SIMD ray to emphasize this fact.

3.1.2 Large Ray Packets

The work of Wald et al. 2001 [76] also demonstrates that tracing multiple rays together offers benefits beyond the extra floating point computation performance afforded by SIMD. Indeed larger ray packets of $n \times n$ with $n = 8$ or $n = 16$ provide up to an order of magnitude improvement over SIMD rays when used for primary visibility. However, the acceleration structure traversal algorithms must change in order to allow for such large packets.

Large ray packets have been demonstrated on kd-trees [57], grids [74], and BVHs [72]. See the recent STAR report [75] for an overview of the build and traversal algorithms for these structures and others. My work focuses on BVHs because they currently exhibit the best combination of build and ray casting performance. However, traversal and frustum culling algorithms are similar between structures, and I believe my results can benefit these other structures as well.

All of these works and their performance numbers target primary visibility, treating point-light shadows as an added bonus. None of them provide in-depth performance comparisons

for ray traced reflections or refractions.

3.1.3 Frustum Bounds for Ray Packets

Tight bounding frusta around coherent ray packets can cull away many ray–Axis-Aligned Bounding Box (AABB) and ray–triangle intersection queries. Frustum culling has been demonstrated for primary rays in Reshetov et al. [57] and for both primary rays and point-light shadows in Wald et al. 2006 [74] and Wald et al. 2007 [72]. To my knowledge, mine is the first ray tracer to use frustum culling for reflection and refraction rays.

3.1.4 Whitted Ray Tracing using Ray Packets

I know of two other works which apply large ray packets to Whitted ray traced effects: Boulos et al. [9] and Mansson et al. [45]. Boulos et al. [9] only study the singular combination of traversal and culling algorithms used in Wald et al. 2007 [72]. Mansson et al. [45] restrict their study to 4×4 ray packets using only the traversal algorithm from Reshetov et al. [57]. I explore a broader range of traversal and culling algorithms and up to 32×32 packets.

Both works demonstrate a 2×–3× hardware performance benefit by using SIMD rays. Boulos et al. [9] only achieve about a 1.5× performance benefit by using large ray packets of 8×8–16×16 over SIMD rays in a Whitted ray tracer. My analysis in Section 3.4 demonstrates that their traversal algorithm can be significantly slower for some scenes when using multiple bounce reflections and refractions. Moreover, I often see 3×–6× performance benefits when using my combination of large ray packet traversal and culling algorithms over SIMD rays.

Mansson et al. [45] explore the possibility of regrouping rays based on various measures of coherence. Their results demonstrate that it is difficult to impossible to efficiently collect more ray coherence beyond what is provided by the image raster. As such, I always group ray packets into $n \times n$ groups as determined by the screen space coordinates of their ancestral camera rays.

3.2 Traversal Algorithms for Large Ray Packets

In this section, I introduce the large ray packet BVH traversal algorithms that I will compare in Section 3.4. I review two existing algorithms, Masked traversal in Subsection 3.2.1 and Ranged traversal in Subsection 3.2.2, and introduce my new algorithm, Partition traversal, in Subsection 3.2.3. Appendix B is provided as a supplement to this section and provides expanded pseudocode for the Ranged and Partition traversal algorithms.

In the descriptions that follow, I use $R = (r_0, r_1, r_2, ..., r_n)$ to denote the set of all SIMD rays in the large ray packet. Using an index i, I can retrieve the ith ray: $R[i] == r_i$. I also differentiate between the concepts of a ray being *active* and *alive*. At any step in the BVH traversal, a ray is *active* if the ray traversal algorithm assumes that the ray overlaps the cell's AABB. A ray is *alive* if it actually does overlap the AABB. All *active* rays are tested against the triangles at the BVH leaves whether or not they are *alive*.

3.2.1 Masked Traversal

I call the first large ray packet algorithm Masked traversal because it uses an array of boolean values to mask out dead rays at the BVH leaves. It is the first and simplest large ray packet algorithm used by Wald et al. 2001 [76] and Reshetov et al. [57] for traversing kd-trees.

The pseudo-code for Masked traversal in Figure 3.2 provides a basis for introducing all of the large ray packet algorithms in this Section. At each step, all rays are tested against the current cell. If *any* ray hits at line 10, then all rays continue through the tree together. At a leaf cell, lines 21– 27, I check if the bounding frustum culls the triangle at line 24, and if not, all alive (unmasked) rays are tested against the triangle. This algorithm can have many extra ray–AABB tests especially deep in the tree.

```
1: // Traverse a Ray packet, R, through theBVH
2: void traverseBVH( Rays R, Frustum F, BVH theBVH )
3:     BVHCell curCell = theBVH.root;
4:     Stack<StackNode> traversalStack;
5:     bool rayMasks[size( R )];
6:     while ( true )
7:       bool anyHit = false ;
8:       if ( frustumIntersectsAABB( F, curCell.AABB() ))
9:         for ( Index i=0; i < size( R ); ++i )
10:          rayMasks[i] = rayIntersectsAABB( R[i], curCell.AABB() ))
11:          if ( rayMasks[i] )
12:            anyHit = true ;
13:            if ( isInner( curCell )) break ;
14:      if ( anyHit )
15:        if ( isInner( curCell ))
16:          StackNode node;
17:          node.cell = curCell.farChild( R );
18:          traversalStack.pushBack( node );
19:          curCell = curCell.nearChild( R );
20:          continue ;
21:        else // isLeaf( curCell ) == true
22:          Triangles T = curCell.triangles();
23:          for ( Index j = 0; j < size( T ); ++j )
24:            if ( frustumIntersectsTriangle( F, T[j] ))
25:              for ( Index i = 0; i < size( R ); ++i )
26:                if ( rayMasks[i] )
27:                  rayIntersectTriangle( R[i], T[j] );
28:      // END if ( anyHit )
29:      if ( traversalStack.empty() )
30:        break ;
31:      StackNode node = traversalStack.pop();
32:      curCell = node.cell;
33:    // END while ( true )...
34:// END void traverseBVH(...
```

Figure 3.2: Pseudo-code for Masked BVH traversal.

3.2.2 Ranged Traversal

Ranged traversal, introduced for use with BVHs in Wald et al. 2007 [72], attempts to avoid many of the ray–AABB tests in Masked traversal by tracking the first alive SIMD ray in R. Let i_a be the index to that SIMD ray. At a BVH cell, I find i_a using the getFirstHit() function:

```
Index getFirstHit( Rays R, Frustum F, AABB B, Index i_a )
    if ( rayIntersectsAABB( R[i_a], B ))  return  i_a;
    if ( !frustumIntersectsAABB( F, B ))  return  size( R );
    for  ( Index i = i_a + 1; i < size( R ); ++i )
      if ( rayIntersectsAABB( R[i], B ))
        return  i;
    return  size( R );
```

A call to getFirstHit() replaces the AABB tests on lines 8– 13 in Figure 3.2.

I track i_a by adding it to the traversal stack's nodes:

```
struct  StackNode
    BVHCell cell;
    Index i_a; // Index to the first alive ray
```

At the BVH leaves, I perform the reverse operation and find i_e, the index to the last alive ray in R:

```
Index getLastHit( Rays R, AABB B, Index i_a )
    for  ( Index i_e = size( R )-1; i_e > i_a; ---i_e )
      if  ( rayIntersectsAABB( R[I[i_e]], B ))
        return  i_e + 1;
```

return $i_a + 1$;

I place a call to getLastHit() right after line 21 in Figure 3.2. All rays in the interval [i_a,i_e) are active and are tested against the triangles in the leaf cell.

By tracking i_a and finding i_e at the leaves, I avoid many ray–AABB intersection tests at the inner cells, but add more ray–triangle intersections at the leaves. For coherent rays, this is acceptable and is a large improvement over Masked, but Ranged traversal can still end up with many extra active rays deep in the BVH leading to potential overhead.

3.2.3 Partition Traversal

My new traversal algorithm, which I call Partition traversal, partitions the rays into strictly alive and dead subsets. My approach is similar to the algorithm in Wald et al. 2007 [73], who aim to increase SIMD utilization for wide (> 4) SIMD units. Distinct from their method, my algorithm is real-time, using an efficient approach to filter out dead rays, and treats the SIMD ray as the smallest ray primitive.

I maintain a separate set of SIMD ray indices, $I = (i_0, i_1, i_2, ..., i_n)$ with $n = \text{size}(R)$. This list is initialized to $I = (0, 1, 2, ..., n)$ at the start of traversal, and instead of tracking the *first* active SIMD ray as I did in Ranged traversal, I use i_a to track one past the last active SIMD ray. I filter out SIMD rays which miss the current BVH cell's AABB with a call to partRays():

```
1: Index partRays( Rays R, Frustum F, AABB B, Indices I, Index i_a )
2:   if ( !frustumIntersectsAABB( F, B ))  return  size( R );
3:   Index i_e = 0  ;
4:   for  ( Index i = 0; i < i_a; ++i )
5:     if  ( rayIntersectsAABB( R[I[i]], B ))
6:       swap( I[i_e++], I[i] );
```

7: **return** i_e;

A call to partRays() replaces the AABB tests on lines 8– 13 in Figure 3.2.

partRays() performs the frustum–AABB test first, then loops through the indices in I, testing each indexed SIMD ray against the cell's AABB. By swapping the elements in I and incrementing i_e at line 6, I is split in-place into two subsets with the indices to the alive SIMD rays in front of i_e. By the end of partRays(), i_e is one past the index to the last alive SIMD ray, and the rest of the SIMD rays indexed by $I[i_e$: size(I)-1] are inactive.

I store the result of partRays() in i_a, and, just as for Ranged traversal, I need to add only this one integer to the traversal stack's nodes. As the ray packet traverses down the tree, the list of alive rays gets smaller. As it pops back up the tree, the SIMD ray ids in I will be re-ordered, but i_a will still point to the end of the alive SIMD ray indices.

In order to test only the alive rays against the triangles at the BVH leaves, I replace the more expensive mask branches in the the loop at lines 25– 27 in Figure 3.2 with a simple indirection:

22: **for** (Index $i = 0$; $i < i_a$; $++i$)

23: rayIntersectTriangle($R[I[i]]$, $T[j]$);

Partition traversal is designed to gracefully handle degradation in ray coherence, and there is nothing limiting the ray packet's size beyond memory bandwidth. However, if the rays in R are truly coherent, then Ranged traversal may avoid more ray–AABB intersection tests.

3.3 Frustum Bounds for Large Ray Packets

In this section, I first review frustum culling basics, and then describe the construction of tight bounding frusta for primary rays in Subsection 3.3.1 and shadow rays in Subsection 3.3.2. I

end with my new approach for bounding reflection and refraction rays in Subsection 3.3.3.

A bounding frustum culls AABBs and triangles using either its 4 bounding corner rays, its 4 side planes, or both. The corner rays cull a triangle when all 4 rays lie outside of the same triangle edge, and they cull an AABB if all 4 rays are separated by the same slab (see Reshetal et al. [57] and Reshetov [56]).

The side planes cull any convex polyhedron (either a triangle or AABB) when all of the polyhedron's vertices lie outside of the same side plane. Let $\vec{n}_i$ and b_i with $0 \leq i < 4$ be the plane normals and offsets for the 4 bounding frustum plane equations, and let p_k be the polyhedron's vertices, then:

$$H_i = \vec{n}_i \cdot \vec{p}_k - b_i. \tag{3.1}$$

If $H_i > 0$ then p_k is outside the plane defined by $(\vec{n}_i, b_i)$, and if all p_k are outside the same plane, then this plane culls the polyhedron. Reshetov et al. [57] show a version of this test optimized for AABBs using SSE instructions.

Given 4 corner rays, I find the frustum's side planes. Let $\vec{o}_i$ and $\vec{d}_i$ with $0 \leq i < 4$ be the corner rays' origins and directions respectively, then:

$$\vec{n}_i = \vec{d}_i \times \vec{d}_{(i+1)\%4} \tag{3.2}$$

$$b_i = \vec{o}_i \cdot \vec{n}_i \tag{3.3}$$

3.3.1 Frustum Bounds for Primary Rays

The task of generating a bounding frustum starts with finding the frustum's 4 corner rays. For primary rays through a pinhole camera, the corner rays are simply the rays at the corners of the $n \times n$ ray packet, and the frustum planes are retrieved directly from Equations 3.2 and 3.3.

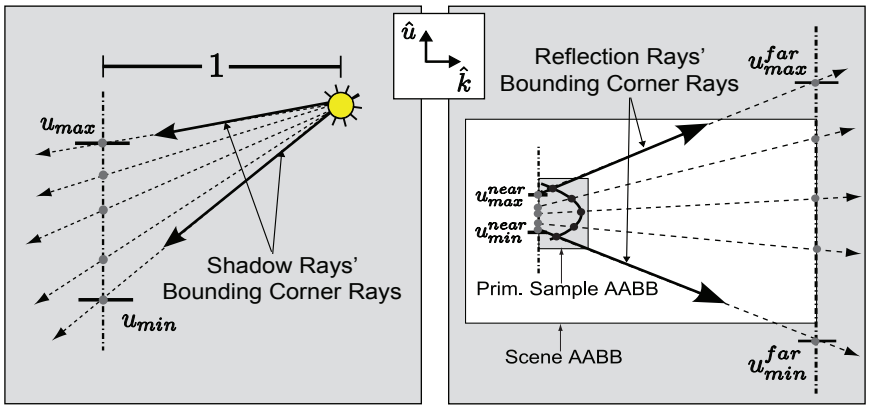


Figure 3.3: Finding bounding frustum corner rays for point-light shadow rays (left) and reflection rays (right).

3.3.2 Frustum Bounds for Point-Light Shadow Rays

For shadow rays, the 4 corner rays defined by the raster are no longer guaranteed to bound the ray volume. Instead, I use an alternate approach as described by Boulos et al. [11] and illustrate this method on the left of Figure 3.3.

I first choose a dominant axis for the ray directions which I call $\hat{k}$. I use the sum $\vec{d^s} = \sum_i^{n \times n} \vec{d_i}$ and take the axis of the max component: $\hat{k} = AxisOf(\max(d^s_x, d^s_y, d^s_z))$. Let the other two axes be $\hat{u}$ and $\hat{v}$. I place an imaginary plane orthogonal to $\hat{k}$ at a distance of 1 in front of the rays' origin. This plane is the vertical dashed line in Figure 3.3. I find the (u, v)–coordinates of the intersection between the ray and the plane which are simply $(u = d_u d_k^{-1}, v = d_v d_k^{-1})$ (I multiply by d_k^{-1} instead of dividing by d_k to emphasize the fact that d_k^{-1} is usually precomputed for each ray for efficient BVH traversal).

Let u_{min} and u_{max} be the minimum and maximum u–coordinates, and accordingly v_{min} and v_{max} the minimum and maximum v–coordinates. The (u, v)–coordinates of directions of the bounding corner rays will then be (u_{min}, v_{min}), (u_{max}, v_{min}), (u_{max}, v_{max}), and (u_{min}, v_{max}), with a 1 or -1 for the k–coordinate, and the origin for all 4 corner rays is simply the location of the point-light. Both the computation of the frustum planes in Equations 3.2 and 3.3 as well as the frustum–AABB and frustum–triangle culling tests based on Equation 3.1

simplify given that the k–coordinate will be 1 or -1, and the common (u, v) values between the neighboring corner ray directions. See Boulos et al. [11] for details.

3.3.3 Frustum Bounds for Reflections and Refractions

The approach in Subsection 3.3.2 only works for rays that meet at a point and so doesn't apply to reflection or refraction rays. Here I introduce a new method which extends to general ray packets, and illustrate my algorithm on the right of Figure 3.3 which shows reflection rays bouncing off of a curved surface.

I start by choosing a dominant axis, $\hat{k}$, exactly as I did in Subsection 3.3.2, but this time, instead of a single imaginary plane, I pick two planes. In order to provide conservative bounds, these planes must bound the paths of all rays in the packet. Therefore, I choose a *far* plane at k^{far} from the scene's AABB in the $+\hat{k}$ direction and a *near* plane at k^{near} in the $-\hat{k}$ direction from the AABB bounding the ray origins. I then find the (u, v)–coordinates of the rays' intersections with both planes, resulting in the intervals $[u^{near}_{min}, u^{near}_{max}]$, $[v^{near}_{min}, v^{near}_{max}]$ at k^{near} and $[u^{far}_{min}, u^{far}_{max}]$, $[v^{far}_{min}, v^{far}_{max}]$ at k^{far}.

The corner ray origins are the extremal intersection points with the near plane: $(u^{near}_{min}, v^{near}_{min}, k^{near})$, $(u^{near}_{max}, v^{near}_{min}, k^{near})$, $(u^{near}_{max}, v^{near}_{max}, k^{near})$, and $(u^{near}_{min}, v^{near}_{max}, k^{near})$. The corner ray directions are the difference between the extremal intersection points with the far plane and these origins. As in Subsection 3.3.2, the frustum planes come from the corner rays using simplified versions of Equations 3.2 and 3.3, and the culling tests are based on simplified extensions of Equation 3.1. My algorithm generates frusta with equivalent characteristics to the frusta used by Reshetov [56] to cull triangles at acceleration structure leaf cells. See Reshetov [56] for details on optimizing construction and intersection tests using this form of frustum bounds.

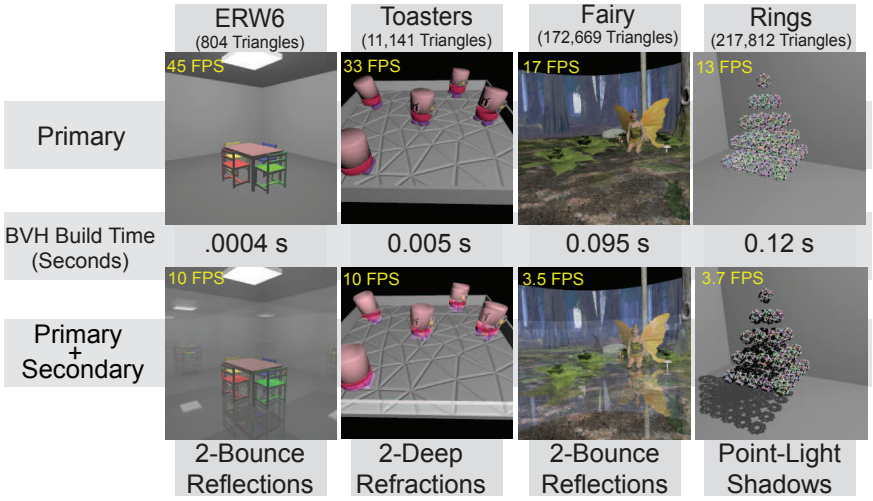


Figure 3.4: The scenes used for evaluating the traversal algorithms from Section 3.2 and the frustum culling algorithms from Section 3.3. All images were rendered at 1024×1024.

3.4 Results–Comparison

In this Section, I analyze the performance characteristics of the ray packet traversal algorithms from Section 3.2 and frustum culling algorithms using frusta generated by the methods from Section 3.3. I first give an overview of the comparison setup in Subsection 3.4.1, I then study traversal algorithms in Subsection 3.4.2 and frustum culling in Subsection 3.4.3. I examine how these algorithms respond to changes in scene complexity, ray recursion complexity, and ray packet size and summarize the results in Subsection 3.4.4. I use the best combination of algorithms to create my real-time Whitted ray tracer in Section 3.5.

3.4.1 Comparison Setup

Hardware Configuration: All tests in this Section (except where otherwise noted) generate images at 1024×1024 resolution on a dual quad-core system (for a total of eight cores) with 2.0GHz Intel Xeon processors. While faster processors and more cores are available, my system is an example of an affordable hardware package. As of the time of writing this chapter, such a system was commonly available for around $2000. The timings include all

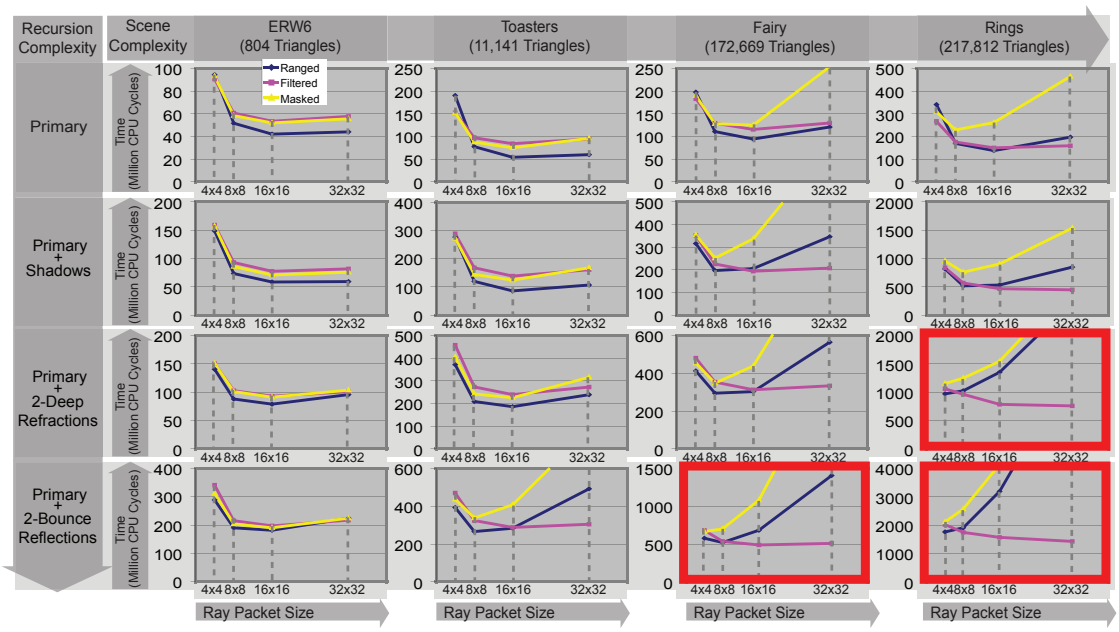


Figure 3.5: Plots of Masked, Ranged, and Partition traversal times for rendering one 1024×1024 image with varying ray packet size, scene complexity, and ray recursion complexity. Masked and Ranged traversal degrade relative to Partition traversal as any of scene complexity, ray recursion complexity, and/or ray packet size increase.

costs related to ray casting and shading. I leave out time to send the image to the graphics card as this adds anywhere from 10% more CPU cycles for the slower renders to 50% for the faster renders. For multi-threaded ray casting, I use the standard approach of tiling the image canvas and dealing out tiles to each thread.

Scenes: I use the scenes shown in Figure 3.4, each of which was chosen for specific qualities. The ERW6 scene is extremely simple, having only 804 triangles and all flat surfaces and presents an ideal environment for a coherent ray packet tracer. The Toasters scene, with 11,141 triangles, targets the lower end of the level of visible complexity in a typical video game scene. The Fairy scene has 172,669 triangles. It is a realistic, high complexity scene, perhaps a future video game scene with both large and tiny objects. The grass at the fairy's knees and the base of the tree is particularly difficult for a packet tracer. The Rings scene from the SPD [28] is specifically intended as a worst case scenario for

reflection and refraction rays. The small and tangled rings serve to disperse secondary rays in all directions.

Acceleration Structure: As previously noted, I use a BVH as the acceleration structure. I use a single-threaded binned SAH build which has previously been shown by Wald [71] to be interactive to real-time even for complex scenes, but is about an order of magnitude slower than state-of-the-art BVH builds using either a grid pre-build as by Wald [71] or a pre-existing scene hierarchy as by Hunt et al. [35] or Yoon et al. [80]. While I focus on ray casting performance, I also include the time to build the BVH from scratch in Figure 3.4 to demonstrate that my acceleration structures are of interactive quality.

Whitted Effects: In order to best evaluate performance for reflection and refraction rays, I set *all* scene materials to be reflective or refractive. This makes for some very difficult situations for my ray packet tracer. The detailed geometry in the Fairy and Rings scenes create some highly incoherent ray packets. I investigate some more reasonable rendering configurations in Section 3.5.

3.4.2 Masked vs. Ranged vs. Partition Traversal

I compare Masked, Ranged, and Partition traversal in the collection of plots in Figure 3.5. Each plot shows time to render one image in millions of CPU cycles (lower values on the y-axis mean faster render times) versus ray packet size. Along the x-axis, I use 4×4, 8×8, 16×16, and 32×32 ray packets. The plots themselves are organized in a table with scene complexity increasing along the x-axis, and ray recursion complexity along the y-axis. From top to bottom, I display ray recursion complexity using primary visibility, primary visibility with point-light shadows, 2-deep refractions, and 2-bounce reflections. I use frustum culling for all results in this Subsection.

Masked traversal, in yellow (or light gray for gray-scale prints), is consistently slower than Ranged traversal, and is slower than Partition in most cases. It was found to work consistently only up to 4×4 ray packets in Wald et al. 2001 [76] and Reshetov et al. [57], and my results agree with these earlier findings. Render times explode with increased packet size for higher scene complexities to the right and deeper levels of ray recursion to the bottom.

As noted in Section 3.2.1, Masked traversal keeps all rays active at the inner cells which can lead to large overheads. If even one packet ray decides to visit a BVH leaf, then all packet rays will be tested against the leaf's AABB. Less coherent packets will have many extra ray–AABB intersection tests leading to poorer performance with increased scene complexity, ray recursion complexity, and packet size.

Ranged traversal, in dark blue (dark gray), behaves significantly better than Masked traversal, and provides the best results on up to 16×16 ray packets for most cases. The most notable exception to this are the three high-lighted plots in the lower right corner with higher scene complexity and higher ray recursion complexity. While Ranged traversal performs much better for the other configurations, a downward trend is clearly visible as I increase scene complexity, ray recursion complexity, and ray packet size.

Partition traversal, in magenta (medium gray), is the most robust to the degrading coherence for higher scene complexity, reflection and refraction rays, and larger ray packets. Both Masked and Ranged traversal reach a breaking point as ray packet size increases. Partition traversal, on the other hand, consistently improves with increased packet size regardless of scene and ray recursion complexity, and is limited only by memory bandwidth and cache coherence which degrades slightly for 32×32 packets.

Partition vs. Ranged traversal: A Closer Look

Ranged traversal and Partition traversal each have their own strengths and weaknesses, and Figure 3.6 shows why. These graphs compare the number of ray–AABB tests and

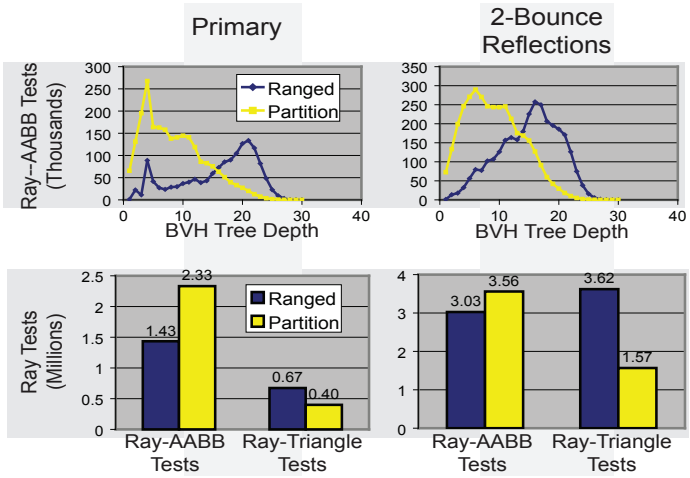


Figure 3.6: Histograms (top) and bar charts (bottom) counting the number of ray–AABB tests and ray–triangle tests required for rendering the Fairy scene at 512×512 using 16×16 ray packets. Ranged traversal is better for primary rays, but Partition traversal is better for reflection rays.

ray–triangle tests because these dominate the ray casting time in my system.

Partition traversal, in yellow (light gray), generates a peak in both histograms higher in the tree, closer to the root, while Ranged traversal, in blue (dark gray), is more peaked at the deeper cells. As described in Subsection 3.2.3, Partition traversal always tests every alive ray against every BVH cell which leads to more ray–AABB tests higher in the tree. Deeper in the tree, most of the rays have been filtered out, so there are fewer ray–AABB tests.

Ranged Traversal, as described in Subsection 3.2.2, can avoid many ray–AABB tests by only testing rays until it finds i_a, the index to the first alive packet ray. The key to success is the probability that most active rays after i_a are also alive deeper in the BVH, particularly at the BVH leaves. This tends to happen for primary visibility resulting in fewer total ray–AABB in the bottom left of Figure 3.6 and faster render times in the top row of Figure 3.5.

However, if the active rays after i_a aren't truly alive, Ranged traversal may suffer big overheads. An incoherent ray packet may avoid some ray–AABB tests higher in the tree

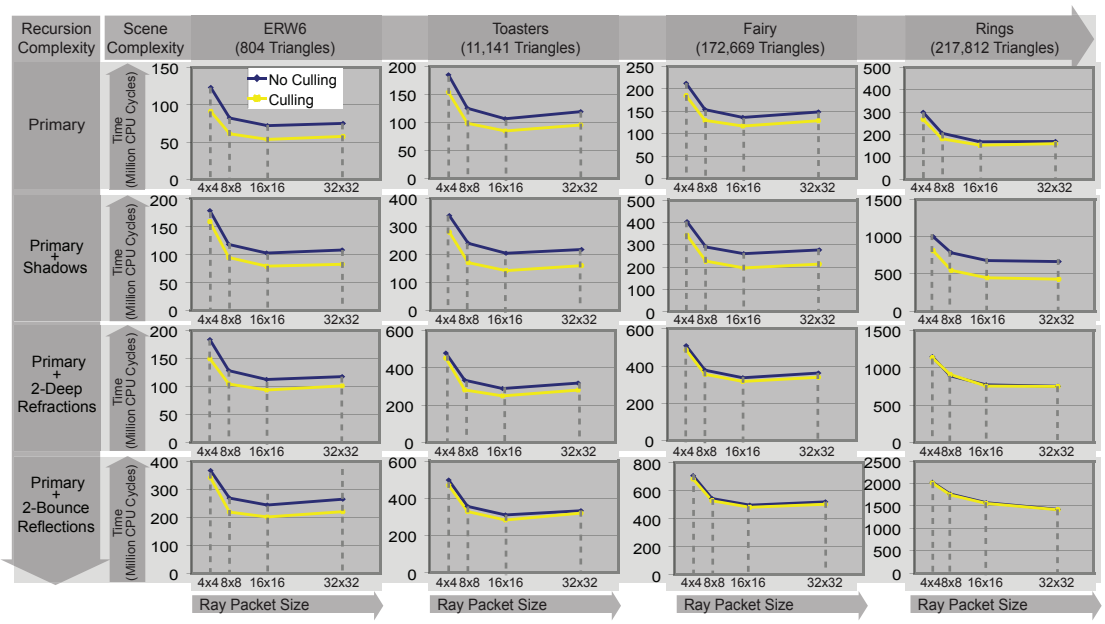


Figure 3.7: Times for rendering one 1024×1024 image with and without frustum culling with varying packet size, scene complexity, and ray recursion complexity. Frustum culling works best for primary rays and point-light shadow rays, and mostly helps reflections and refractions off of flat and smooth surfaces.

only to have to perform them deep in the tree where there are exponentially more BVH cells, causing the higher peak at deeper BVH cells in the histograms in Figure 3.6. Even worse, if dead rays reach the leaves, there will be many more expensive ray–triangle tests as shown at the bottom of Figure 3.6. For the more coherent primary ray packets on the left, these extra ray–triangle tests are acceptable since ray–AABB tests dominate ray casting time, but the less coherent secondary ray packets on the right lead to many extra ray–triangle tests and slower render times in the highlighted plots in Figure 3.5.

3.4.3 Frustum Culling for Whitted Ray Tracing

In this Subsection, I evaluate frustum culling for all ray types in a Whitted ray tracer using the algorithms in Section 3.3 to construct tight bounding frusta. All results in this Subsection were generated using Partition traversal since this presents the most stable baseline.

Figure 3.7 demonstrates the benefits of frustum culling. As in Subsection 3.4.2, each

plot shows time to render one image versus ray packet size, and the plots are organized left to right in order of increasing scene complexity and top to bottom in order of increasing ray recursion complexity.

In general, frustum culling works best for primary visibility and point-light shadows (the top two rows in Figure 3.7). There is some benefit for frustum culling on reflection and refraction rays for the relatively simple ERW6 and Toasters scene, but barely any noticeable benefit for the more complex Fairy and Rings scene. We will see in Section 3.5 that the results for the Toasters scene is more representative of most rendering configurations where not all surfaces are reflecting and/or refracting.

We expect good results for primary rays from a pinhole camera, but it is less clear why culling for point-light shadow rays is more effective than for reflection and refraction rays. Point-light shadow rays converge at the light source which leads to tighter ray packets and hence tighter ray bounds. Reflections and refractions, on the other hand, tend to diverge making it significantly more difficult to generate tight bounding frusta.

3.4.4 Packet Traversal and Frustum Culling: Summary

I summarize my conclusions for ray packet traversal and frustum culling in Table 3.1. Masked traversal is superseded by Ranged and Partition traversal. Ranged traversal is the best for primary visibility on packets of up to 16×16. For secondary rays, Ranged traversal tends to be the best on packets of up to 8×8, but runs the risk of falling to the pressure of increased scene and ray recursion complexity. Partition traversal should be used for secondary rays in systems that require large ray packets, complex scenes, or where ray–geometry intersection tests dominate ray casting time. Alternatively, Partition traversal can be used for *all* secondary rays as a conservative measure to avoid the pitfalls of Ranged traversal.

Frustum culling works well for primary visibility and point-light shadows providing up to 2× benefit. The results for reflection and refraction rays are less impressive, speeding

up ray casting times mostly for relatively flat and smooth surfaces. Frustum culling does help, generally by about 1.2×–1.3×, but it should not be relied upon to achieve real-time performance.

3.5 Results–Performance

In this Section, I construct my real-time Whitted ray tracer and evaluate its performance. I find that the analysis in Section 3.4 leads to robust real-time performance, and large ray packets offer significant benefits over SIMD rays for Whitted ray tracing.

Based on the recommendations from Section 3.4 and Table 3.1, my ray tracer uses 16×16 ray packets with Ranged traversal for primary rays, and I choose between Partition and Ranged traversal for reflection, refraction, and shadow rays based on the scene and ray recursion complexity. I use frustum culling for all ray packets. All hardware configurations used to generate results in this Section are the same as in Section 3.4.

Figure 3.1 shows several scenes and ray recursion configurations rendered with my real-time Whitted ray tracer. These images tend to render significantly faster than those from Section 3.4 because only select surfaces are set to be reflective or refractive. The Toasters scene is the same scene from Section 3.4, but I have set 1-bounce reflections for the floor and turned on point-light shadows. The Fairy scene is also used in Section 3.4, but I have set 1-deep refractions on the wings, and 1-bounce reflections on the forest floor making it appear as if the fairy is sitting on water. While it isn't particularly noticeable from this view, reflections are turned on for the fairy's eyes as well as the gold and jewels on her wand.

Figure 3.1 also includes two scenes from the BART [42] collection. I render only the first keyframe from these sequences. The museum image demonstrates deep reflections with 3-bounces. The kitchen scene uses 1-bounce reflections, 4-deep refractions, and point-light shadows from one point-light. Notice the refractions through the bowls and glasses on the table as well as the dragon model under the table. Light even refracts through the dragon's

	Toasters	Fairy	BART Museum	BART Kitchen
2×2 SIMD Ray Packets	1.9 FPS	2.1 FPS	2.4 FPS	1.2 FPS
16×16 Ray Packets	11.8 FPS	6.7 FPS	8.5 FPS	4 FPS
Performance Benefit	**6.1** $\times$	**3.2**$\times$	**3.5**$\times$	**3.3**$\times$
Reflect+Refract Traversal	Ranged	Partition	Partition	Partition
Culling Benefit	1.35$\times$	1.19$\times$	1.18$\times$	1.19$\times$

Table 3.2: Comparison of my large ray packet tracer using 16×16 packets against 2×2 SIMD ray packets for rendering the images in Figure 3.1.

reflection.

I compare to 2×2 SIMD ray packets in Table 3.2. This table presents times for rendering the images in Figure 3.1 for SIMD rays and my large ray packet tracer and whether secondary rays use Partition or Ranged traversal. In all examples, large ray packets provide at least 3$\times$ faster render times over SIMD Rays and up to 6$\times$ for the simpler Toaster scene.

In the last row of Table 3.2, I include the performance benefit due solely to frustum culling. For these configurations, frustum culling improves performance by about 18%–35% which is significantly more than reported in Section 3.4 for the Fairy and Rings scene with reflection and refraction rays. As is usually the case in Whitted ray traced scenes, the reflective and refractive surfaces used in this Section tend to be significantly flatter and smoother leading to better culling performance.

3.6 Conclusion

This chapter introduces a fully real-time CPU-based Whitted large ray packet tracer. Entering this new domain required serious analysis of large ray packet algorithms for scene traversal and frustum culling. It also required the new Partition traversal algorithm and a new approach for generating frustum bounds around reflection and refraction rays. The result is a real-time

Whitted large ray packet tracing system which is robust to degrading coherence.

I thoroughly evaluated ray packet algorithms for frustum culling and three BVH traversal algorithms in the context of real-time Whitted ray tracing. There are a large number of possible combinations of these algorithms, and in the process of this work, I evaluated many of them which are not presented. I found that the simple solutions work best and believe the algorithms presented here most concisely encompass the results of my research.

Distributed ray traced effects are also likely to benefit from my work. Here I focus on Whitted ray traced effects to push them into real-time, but real-time results remain out of reach for distributed ray tracing. Based on the results in this chapter, I believe this class of effects requires new algorithms beyond ray coherence based techniques to join the interactive domain.

Chapter 4

Adaptive Wavelet Rendering for High-Dimensional Effects

4.1 Introduction

The previous chapters focus on accelerating specific light transport effects. Chapter 2 focuses on soft shadows from rectangular light sources, and Chapter 3 targets reflections and refractions. However, photorealism requires many other effects and general combinations of them.

Rendering photorealistic images with effects such as depth of field, area lighting, motion blur and global illumination requires the evaluation of a complex high-dimensional integral at every pixel. Each effect adds one or more dimensions to the integral, and each dimension adds another potential source of variance. Monte Carlo integration is a robust approach for estimating this integral, but requires many samples to reduce variance to tolerable levels. The beam tracer from Chapter 2 can be used in lieu of Monte Carlo integration for relatively simple integrals, such as soft shadows from polygonal area light sources. However, it is significantly easier to solve general high-dimensional integrals with the point samples from a ray tracer rather than the area samples from a beam tracer. The packet ray tracing algorithms

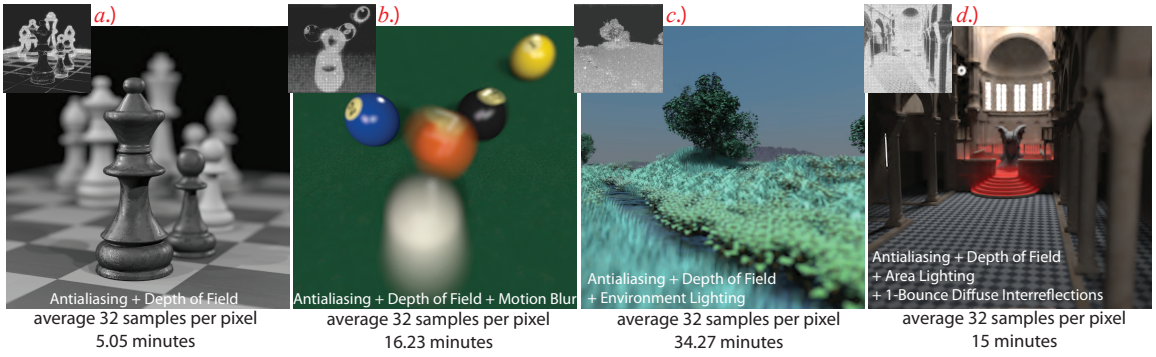


Figure 4.1: All images were rendered at 1024×1024 on a single core of a 2.8GHz Core2 Extreme laptop. By adaptively sampling and reconstructing in a smooth wavelet basis, we get near-reference quality noise-free images with only 32 samples per pixel, with general high-dimensional combinations of rendering effects. The insets show the Monte Carlo sample distributions that generated the images.

from Chapter 3 can be used to accelerate Monte Carlo integration, but the number of rays required can still make brute force integration prohibitive. Fortunately, natural images have smooth regions either in the image domain, over the other dimensions, or both. We should therefore adapt to this smoothness rather than performing an exhaustive sampling.

However, most adaptive algorithms sparsely sample *either* smoothly varying image regions, or slow variation in other dimensions, but not both. This leads to noise and artifacts at low sample counts, as seen in Figure 4.2. A recent adaptive multidimensional sampling method [27] addresses these issues, but it scales poorly to general higher-dimensional integrals involving multiple effects. Even in low-dimensional situations, there can be computational and memory overheads in both its sampling and reconstruction stages that are particularly costly for recent optimized ray-tracers [76, 57].

I propose *adaptive wavelet rendering* to directly estimate the image in the wavelet domain, and thus robustly handle all forms of variance. As opposed to pixels, wavelets present a multi-scale view of the image, and so provide a good representation for both image edges *and* smooth image features [44, 66]. This characteristic has made wavelets one of the most popular formats for image and video compression, and also for accelerating finite element methods such as radiosity [23] and PRT [49]. Despite their proven benefits

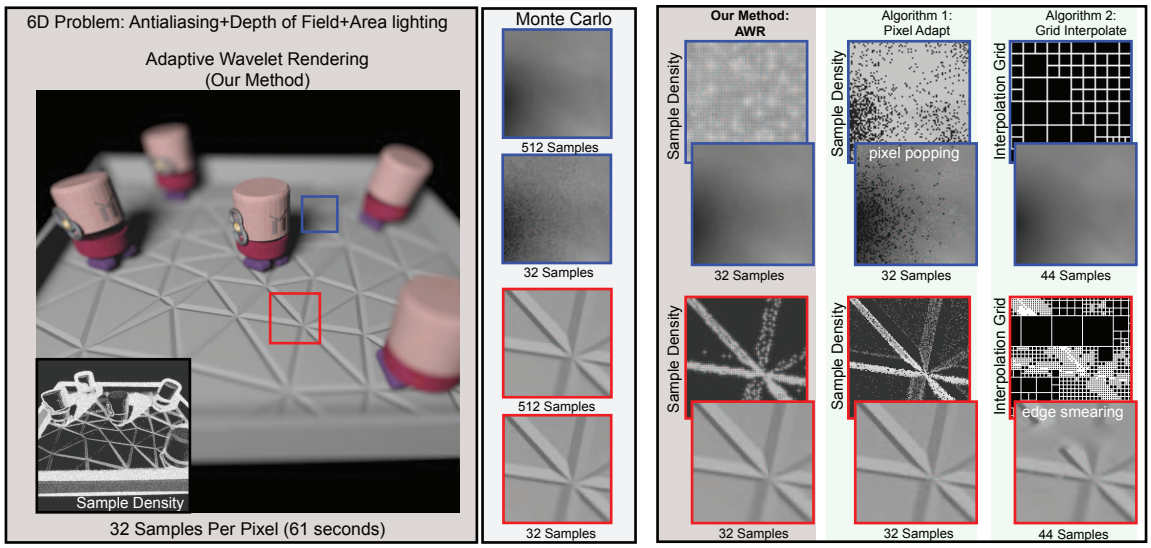


Figure 4.2: Each pixel in this image is a 6 dimensional integral (2D image-space for antialiasing, 2D lens for depth of field, and 2D for an area light). My method computes a reference-quality image using an average of only 32 samples per pixel. To the right are close-ups along with sample distributions generated using 4 different algorithms. The top row shows a smooth region of the image that has high variance from the other integral dimensions. The bottom row shows image-space edges that are smooth over the integral dimensions. My method performs well in both regions. The pixel adaptive algorithm 1 has considerable noise in the smooth image areas on top, due to variance in the integral dimensions. The grid interpolation algorithm 2 has artifacts in the bottom row at edges, and it must exhaustively sample the integral dimensions. In the sample density images, note that adaptive wavelet rendering gives samples both to image-space edges and regions that are smooth in the image but have high variance in other integral dimensions. Samples also cluster at nodal points for wavelet interpolation.

elsewhere, wavelets have rarely been used to speed up Monte Carlo sampling (with the notable exception of Bolin and Meyer [8]), and I am inspired by recent work that shows their benefit for importance sampling [13].

Because the wavelet reconstruction of the image is hierarchical, coarse-scale wavelets are better at reconstructing large, smooth regions of the image, whereas finer-scale wavelets resolve small details, such as detailed texture and edges. I exploit this property in my algorithm to obtain an optimal hierarchical sample distribution.

As I describe in Section 4.4, my algorithm is composed of two simple stages: adaptive sampling and image reconstruction. In the first stage, I iteratively measure the variance of

the wavelet basis' scale coefficients. Since high frequency details cause high amplitude wavelet coefficients, I use the wavelet magnitude to locate small-scale features, such as edges. The algorithm further samples those coarse-level scale coefficients that have high variance but do not have high wavelet magnitudes, i.e., do not have strong edges. These are image regions with high variance from the other dimensions. Finer scale coefficients receive samples to resolve the remaining high-frequency image features. New samples are drawn from the coefficients' scale function via importance sampling to reconstruct a smooth image in the reconstruction stage. Hence, my algorithm naturally adapts to both image-space edges as well as smooth regions with variance from the other integral dimensions.

In the second stage (image reconstruction), I use the wavelet basis to smooth away any remaining noise. I consider *all* of the wavelets (as opposed to standard truncation of small values), and simply subtract the measured variance from the wavelet coefficient magnitudes. This is conceptually similar to choosing the smoothest image that fits the measured statistics. In smooth regions, this allows the adaptive sampler to send more samples to the coarser scale functions, effectively sampling the image at a lower resolution. Thus, the adaptive sampling stage cooperates with the image reconstruction to efficiently compute a relatively noise-free image even with minimal sample budgets.

Adaptive wavelet rendering has the following key features:

Low Sample Counts: As seen in Figure 4.1, my results are relatively free of noise with an average of only 32 samples per pixel. In fact, because of the variance-reducing image reconstruction stage, visually consistent results are usually obtained even for 16 samples per pixel (see Figure 4.9).

Efficiency: My algorithm has low computational and memory overheads. Moreover, it is conceptually simple and easy to implement. I have implemented it within a SIMD optimized packet ray tracer, and have found it to perform significantly faster than standard Monte Carlo path tracing [38] and multidimensional adaptive sampling [27]. All of the images in Figure 4.1 were rendered in a matter of minutes, and Figure 4.2 in only 61

seconds, both using a single core on a 2.8GHz Core2 Extreme processor. Images such as these often take several hours to generate using traditional methods.

Generality: The wavelet representation is only over the 2D image domain, and my algorithm directly considers only image-space values and variance. Thus, the method handles general combinations of effects, and does not suffer from the curse of dimensionality. Note that the scene in Figure 4.1*d* includes antialiasing, depth of field, area lighting, and diffuse global illumination for an 8D integral. My method is most powerful when used to simulate effects that produce a smooth result, such as depth of field, which are particularly difficult for Monte Carlo algorithms.

4.2 Previous Work

4.2.1 Parametric Integration and Curse of Dimension

For solving a single integral, Bahvalov's theorems (see Haber [26] which references Bahvalov [5]) state that the best-case performance benefit of any numerical integration algorithm over standard Monte Carlo decreases exponentially with the number of dimensions. Fortunately, rendering is an instance of "parametric integration" with many correlated integrals. Based on this insight, Keller [39] builds on the work of Heinrich and Sindambiwe [34] to develop a multi-level Monte Carlo algorithm, with interpolation used to solve multiple integrals at once and make up for the curse of dimensionality. However, they note artifacts, such as smearing across discontinuities. I also choose to focus on image-space interpolation, rather than chase diminishing returns in the integral dimensions. My work differs in that I include the image-space dimensions in both the parametric interpolation problem and the integral, and I use wavelets to distinguish image-space discontinuities from smooth variation.

Multidimensional Adaptive Sampling: The recent multidimensional adaptive algorithm in Hachisuka et al. [27] also takes advantage of smoothness in the parametric image-space dimensions to produces high-quality images with very low sample densities. However, it is affected by the curse of dimensionality in two places. The adaptive sampling portion of the algorithm provides diminishing returns as the dimensionality increases (as predicted by Bahvalov's theorems) and the computational cost of the signal reconstruction stage is exponential in the dimensionality. I also find that this method introduces blocky artifacts for higher dimensional problems (see Figure 4.7*o* and *t*). As such, this solution is effective mostly for low dimensional problems with expensive shading costs ($d <= 4$).

4.2.2 Basic Adaptive Techniques

Most traditional adaptive sampling approaches fall into one of two categories. Some rely on a purely local measure of variance, usually within a single pixel, to adaptively determine the number of samples for the Monte Carlo integral [78, 48]. This works well for edges, but tends to provide uneven samples over smooth regions and so either generates artifacts or requires many more samples to reproduce a smooth result. Alternatively, algorithms in the second category exhaustively sample the integral at specific points, often the vertices of a grid. They then attempt to interpolate between these nodal points to reconstruct smooth features, while locating high-frequency image-plane regions to focus more samples upon. This approach can smear discontinuities or fail to locate small features. More recent advances [25, 6] better locate edges and other key image features but still must oversample the nodal points, and so do not take advantage of regions of low variance.

Figure 4.2 shows examples of these algorithms. Algorithm 1 is a simple example of the first category, while Algorithm 2 is a simple example of the second. The sample distributions depicted to the right are characteristic of such algorithms, and so are the corresponding artifacts. My algorithm's distribution exhibits the best qualities of the two strategies. Similar to the first category, it spreads samples across the image, which is best

for finding edges. More like the second category, the samples cluster at nodal points which are used to interpolate smooth results.

Veach and Guibas [70] apply a variant of Metropolis Monte Carlo to simulating light transport. This algorithm is intended for rare event simulation to bring out highly focused local effects such as caustics or indirect light leaking through a small opening. It may be best to use my adaptive sampler for a baseline sampling, then Metropolis to capture the rare events, and lastly my wavelet reconstruction to remove the noise.

Perceptually Based Adaptive Sampling: Of particular note is the work of Bolin and Meyer [8] who develop a sophisticated visual error metric and use it for adaptive sampling. Their work emphasizes that adapting to variance in a multiscale wavelet hierarchy corresponds more closely to the human visual system. However, their sampling algorithm still resorts to distributing samples to the leaves of the wavelet hierarchy and so is in a similar category as Algorithm 1 above. In smooth regions, my wavelet reconstruction removes the error at the finer wavelet levels, so I can send more samples to the coarse level scale coefficients, effectively sampling at a lower resolution while more accurately capturing smooth effects. Moreover, their sampling algorithm does not work for wavelets with overlapping support, and so can only use the Haar wavelet basis. The sampling stage of my algorithm is an efficient framework for working with arbitrary discrete wavelet bases, such as the smoother Daubechies 9/7 and LeGall 5/3. Note that Bolin and Meyer's advanced visual error metric can be used in place of the simple contrast metric in Section 4.4.1

Multidimensional Lightcuts: A recent method is Multidimensional Lightcuts (MDLC) [77]. Their method is not an adaptive sampling approach in the same sense as my work and so is not directly comparable. For input, MDLC takes a constant number of primary shade points (gather points) and a constant number of light points. MDLC is an elegant approach for reducing the number of gather-point vs. light-point pairings. However, it does not adapt the number of gather-points or light-points, and so must start with an oversampling of both to

guarantee high-quality convergence. Also, this method only considers gather-points within a pixel, and does not share information between neighboring pixels. As such, it may be best to combine their approach with mine. MDLC may help in situations where there are many spatially coherent light sources, while my algorithm can be used to adaptively introduce gather-points and interpolate the regions of smooth variation.

4.2.3 Adaptive Noise Removal

There has been significant work in adaptive post-production noise reduction filters. Wavelets are commonly used for noise reduction, using either hard thresholding or soft thresholding on the fine-scale wavelets [66]. Hard thresholding simply clamps wavelets to a low value. Soft thresholding subtracts a constant value from the wavelet magnitudes. My wavelet reconstruction improves on soft thresholding by subtracting a measure of the wavelet variance from the wavelet magnitude.

Besides wavelets, other bases may be used. The work of Meyer and Anderson [47], for example, removes noise in animated sequences by projecting the image sequence onto a compressed PCA basis.

Two other directions of research derive from anisotropic diffusion introduced by Perona and Malik [53] and bilateral filtering from Tomasi and Manduchi [69]. Anisotropic diffusion is an iterative approach, and as such may be subject to instabilities. Moreover, it is often slower than either bilateral filtering or my reconstruction stage. Solutions based on bilateral filtering often suffer from objectionable ringing around image edges.

All of these methods focus on image reconstruction alone. Despite significant research dedicated to preserving image features [46, 79, 58], it remains difficult for standalone post-processing noise removal algorithms to distinguish features from noise. This is because there is often simply not enough statistical information at the pixels to construct an accurate result.

My work demonstrates the benefit of connecting adaptive sampling to reconstruction, and

tailoring the adaptive sampling algorithm to the specific attributes of the reconstruction stage. By doing so, I am able to sample large smooth regions at an effectively lower resolution (see closeups in the top row of the right side of Figure 4.2) while also sending more focused samples to edges and other discontinuous image features (see bottom row of the right side of Figure 4.2).

4.2.4 Interactive Ray-Tracing

The works of Wald et al. [76] and Reshetov et al. [57] exploit coherence between ray samples to amortize ray-tracing and shading costs across packets of 16–256 rays and achieve interactive rates on simple scenes. Although the rays for multidimensional effects are incoherent (see Boulos et al. [9] and Overbeck et al. [52]), brute force rendering with an optimized ray-tracer is in some cases as effective as an expensive adaptive technique (see right two columns of Figure 4.7). In light of such benefits, I believe a valuable aspect of my algorithm is that it allows for use with a packet ray-tracer, since I adapt to image regions rather than only individual pixels. In addition, my method has low overhead even when used with a highly optimized ray-tracer.

4.2.5 Frequency Analysis

Recent work has also studied light transport in the frequency domain [18, 19, 64], leading to simple image-space sampling heuristics based on local frequency content. My work is in some ways the logical extreme of this approach, creating the image directly in the wavelet basis. Note however that general phenomena can be addressed, without needing to analyze or compute multidimensional space-angle Fourier spectra. Moreover, by using wavelets I can better localize both spatial (edges) and low-frequency (smooth) effects simultaneously.

4.3 Background: Wavelets and the DWT

Before describing the details of my algorithm, I provide a brief introduction to wavelets and the discrete wavelet transform (DWT).

A 1D wavelet basis is defined by the translates and dilates of a scale basis function ϕ, and a wavelet basis function ψ. Following the JPEG 2000 image compression standard [61], I use Daubechies 9/7 wavelets [14] (also referred to as Cohen-Daubechies-Feauveau wavelets) for all examples in this chapter, and I also found LeGall 5/3 wavelets to be effective. Their precise forms are given in Appendix C.1, and profiles of their analysis scale functions are shown later in Figure 4.5.

The entire 2D wavelet basis is defined as

$$\begin{aligned} \Phi_{k,ij}(x,y) &= \Phi(2^{-k}x - i, 2^{-k}y - j), \\ \Psi^{\alpha}_{k,ij}(x,y) &= \Psi^{\alpha}(2^{-k}x - i, 2^{-k}y - j), \end{aligned} \quad \text{with} \begin{cases} 0 \le \alpha < 3, \\ 1 \le k \le n, \\ 0 \le i < i_k, \\ 0 \le j < j_k. \end{cases}$$

In this expression, k indexes the "level" of the wavelet, with $k = n$ the coarsest or most dilated level, and $k = 1$ the finest level. I reserve $k = 0$ to refer to the original pixels. I use i and j for the translations, with $i_k = j_k = 2^{n-k}$ for square images of size 2^n.

The process of transforming a pixel image into a multi-scale wavelet basis is called *analysis*, and the inverse process back to pixels is called *synthesis*. The most general form of wavelet analysis computes the inner product between image $B(x, y)$ and wavelets:

$$S_{k,ij} = \left\langle B, \Phi_{k,ij} \right\rangle = \int\int B \cdot \Phi_{k,ij} dxdy, \tag{4.1}$$

$$W^{\alpha}_{k,ij} = \left\langle B, \Psi^{\alpha}_{k,ij} \right\rangle = \int\int B \cdot \Psi^{\alpha}_{k,ij} dxdy. \tag{4.2}$$

The $S_{k,ij}$ are referred to as *scale coefficients* and the $W^{\alpha}_{k,ij}$ are the *wavelet coefficients*.

In practice, I perform a discrete wavelet transform (DWT), for both analysis and synthesis, with cost linear in the number of pixels. For both analysis and synthesis, the DWT applies two filters, one low-pass and one high-pass, which together form a "filter bank". I use the so-called "non-standard" DWT, where I alternate between applying the 1D DWT on the image rows, and then on the columns. In my application I use the DWT for its efficiency, but for the rest of the chapter, I will be using the inner product form in Equations 4.1 and 4.2 for notational convenience.

4.4 Wavelet Rendering Algorithm

Rendering a single pixel with anti-aliasing and high-dimensional effects requires the evaluation of an integral at every pixel:

$$B(x,y) = \int_{y}^{y+1} \int_{x}^{x+1} \int_{\mathbf{s}} F(u,v,\mathbf{s}) d\mathbf{s} du dv, \tag{4.3}$$

where $\mathbf{s}$ compactly denotes all the high-dimensional effects, such as depth of field (lens aperture), motion blur (time), and/or soft shadows (area light). The function F is evaluated by Monte Carlo sampling, and is treated as a black box by my method. The goal is to compute all $B(x,y)$ using as few samples, i.e., evaluations of F, as possible.

The adaptive sampling portion of my algorithm (Section 4.4.1) tightly cooperates with the reconstruction stage (Section 4.4.2) to produce a smooth result. Both stages use the wavelet basis to identify high variance regions of the integral. So rather than directly computing Equation 4.3, my algorithm keeps track of the scale and wavelet coefficients in the wavelet basis. To compute the scale coefficients, I estimate:

$$S_{k,ij} = \left\langle \int F d\mathbf{s}, \Phi_{k,ij} \right\rangle = \int \int \int F \cdot \Phi_{k,ij} d\mathbf{s} dx dy. \tag{4.4}$$

The adaptive sampler iteratively distributes samples to achieve a low variance estimate of

Equation 4.4 for all scale coefficients. The reconstruction stage then subtracts the remaining estimated error from the wavelet coefficients to synthesize a smooth image.

4.4.1 Adaptive Wavelet Sampling

The key to the adaptive sampling process is determining which scale coefficient receives new samples at each iteration. Coarse-scale high variance image features (like the blur from an out-of-focus lens) will cause high variance at all levels in the wavelet hierarchy. However, if the final result is smooth, then the scale functions at the coarser levels may predict a more accurate result than the noisy wavelet functions at the finer levels. In these situations, we would like the adaptive sampler to compute an accurate result for the coarse scale coefficients, and rely on the reconstruction phase to remove the noisy wavelets. On the other hand, to handle more isolated fine-scale image features like edges, we prefer a more focused sample distribution.

Definitions: Before I detail my approach to meeting the above requirements, I must first define the variables I use to compute image variance and wavelet coefficients. I accumulate the Monte Carlo samples $F(u, v, \mathbf{s})$ at the image pixels, and maintain estimates of the pixel mean $\widetilde{B}$ and the variance of these samples, $\sigma^2(F)$. There are many methods to approximate the functional variance from a set of samples. I use the square of the contrast metric used by Mitchell [48]:

$$\sigma^2(F) = (I_{\max} - I_{\min})^2/(I_{\max} + I_{\min})^2, \tag{4.5}$$

where $I_{\max}$ and $I_{\min}$ are the maximum and minimum sample intensity respectively. The numerator provides an upper bound estimate of the pixel variance, and the denominator weights the adaptive sampler towards the darker image regions, where the human eye is more sensitive to error. Given these definitions, the estimator variance of the pixel mean is

$$\sigma^2(\widetilde{B}) = N^{-1}\sigma^2(F), \tag{4.6}$$

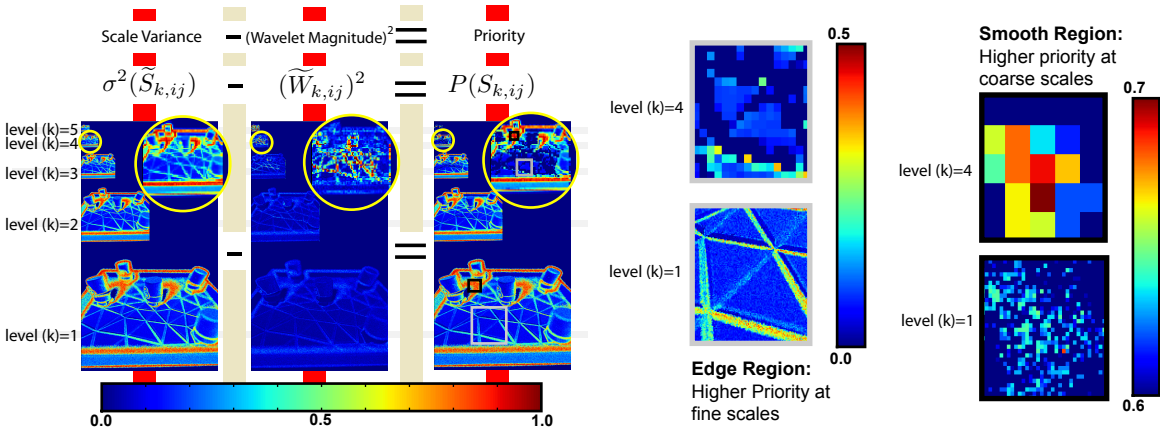


Figure 4.3: The process of computing Equation 4.8, for the priority values of different scale coefficients. Smooth regions have higher priority at coarse scales. Edges have higher priority at finer scales.

Step 1—Initialization: Coarsely sample the entire image, inserting scale coefficients at levels $0 \leq k \leq 5$ into a priority queue.

Step 2—Priority Computation: Update the priority of each scale coefficient in the queue.

Step 3—Sampling: Pop the next scale coefficient from the priority queue and importance sample it.

Step 4—Iterate: If samples remain, Goto 2, else finish.

Figure 4.4: Steps of the adaptive sampling stage. My algorithm starts with a fixed sample budget, and iteratively distributes samples to high variance scale coefficients until the samples are depleted.

where $N(x, y)$ is the number of samples that land in pixel (x, y). I later show how to compute the variance of the scale coefficients using this information. The wavelet coefficients are computed simply by performing a DWT analysis on the pixel means:

$$\widetilde{W}^{\alpha}_{k,ij} = \left\langle \widetilde{B}, \Psi^{\alpha}_{k,ij} \right\rangle. \tag{4.7}$$

The Algorithm: Now, given a fixed budget of samples, the adaptive sampling stage proceeds according to the steps in Figure 4.4.

Step 1—Initialization: I generally start by coarsely sampling the image with 4 samples per pixel. I now insert scale coefficients from levels $0 \leq k \leq 5$ (including the pixels, which we may recall are the finest-level scale functions) into the priority queue.

Step 2—Priority Computation: The priority function or oracle $P(S_{k,ij})$ determines which coefficients require more samples. A number of oracles have been proposed in other contexts, for instance for refinement of links in wavelet radiosity [23]. My heuristic is designed to send more samples to coarse scale coefficients in smooth regions of high variance, and so it prioritizes coefficients that have a large functional variance over their support. However, when a high-frequency and non-oscillating image feature like an edge exists, it is better to allocate samples to finer scales, where the edge is resolved clearly.

Since the wavelet coefficients are (by definition) equal to the error due to image edges and other high-frequency features, simply subtracting the wavelet magnitudes from the scale variance results in the desired heuristic:

$$\boxed{P(S_{k,ij}) = \sigma^2(\widetilde{S}_{k,ij}) - (\widetilde{W}_{k,ij})^2} \tag{4.8}$$

I use the average of the 3 wavelets squared for the wavelet magnitude: $(\widetilde{W})^2 = \frac{1}{3}\sum_\alpha (\widetilde{W}^\alpha)^2$. To estimate $\sigma^2(\widetilde{S}_{k,ij})$, I perform a wavelet analysis on the per-pixel variance, using the square of the scale function:

$$\sigma^2(\widetilde{S}_{k,ij}) \approx \left\langle \sigma^2(\widetilde{B}), \Phi^2_{k,ij} \right\rangle. \tag{4.9}$$

I derive this equation in Appendix C.2. The inner product in Equation 4.9 is an unbiased estimator (as long as the pixel samples are uncorrelated). Since the DWT is more efficient to compute, I perform a DWT analysis on the per-pixel variances ($\sigma^2(\widetilde{B})$) using the squares of the filter bank coefficients. This biases my estimate of $\sigma^2(\widetilde{S}_{k,ij})$ when using filters with overlapping support, but it works well in practice. Note that this bias only affects the priority values in Equation 4.8 and not the final result. For efficiency, I only use the low-pass portion

of the filter bank, since we only require the resulting scale coefficients.

Discussion: The heuristic in Equation 4.8 distinguishes between two types of image-space variance: 1) smooth regions of high variance (like the blur from an out-of-focus lens or the penumbra of a soft shadow), and 2) edges and other nonsmooth image features. For the smooth regions, we should send more samples to the coarse scale functions, and my wavelet reconstruction will interpolate the result across the smooth region. For edge regions, we need more focused samples to resolve the feature, so samples should go to the finer scale functions. In the far left image in Figure 4.3, observe that for smooth regions (like the penumbra from one of the toasters' shadows), the scale variance increases from fine scale to coarse scale. Thus, the scale variance correctly prioritizes these regions. For edge regions (like the edges on the floor), the variance is about the same across levels, and so will not necessarily target the finer scale functions. However, observe that the values of the squared wavelet coefficient magnitudes (the second image from the left in Figure 4.3) tend to grow from fine to coarse for edge regions, and stay the same across levels for smooth regions. Thus by simply subtracting the squared wavelet magnitudes from the scale variance, as in Equation 4.8, we achieve a heuristic which correctly handles both regions.

The images on the right of Figure 4.3 are close-ups of an edge region and a smooth region from the priority image. I have rescaled the color maps to highlight the different priority values between levels. Note that the priorities for the smooth region are higher at the coarser level 4 than level 1. Alternatively, the priorities for the edges are higher at the finer level 1 than level 4.

Implementation Details: The estimator in Equation 4.9 can be tuned to target samples toward coarser or finer levels by renormalizing the $\Phi^2_{k,ij}$ filters to sum to a value > 1. I found empirically that renormalizing to 1.05 works well in most situations and tends to target levels $0 \leq k \leq 4$.

Computing Equation 4.8 requires 2 DWT analyses: one standard analysis for the wavelet coefficients $\widetilde{W}^{\alpha}_{k,ij}$, and one with the squared low-pass filter for $\sigma^2(\widetilde{S}_{k,ij})$. To make these

analyses efficient, I perform them locally over an affected image area as new samples are added. This requires that I keep track of some amount of information that is usually discarded during a wavelet transform. Specifically, I must keep all scale coefficients, and the intermediate wavelet coefficients that result from applying the non-standard 2D DWT in one dimension before applying it in the other dimension. Despite this extra memory requirement, the performance benefit from updating only the affected coefficients greatly outweighs this minor expense.

Step 3—Sampling: In Step 3, the highest priority scale coefficient is taken from the priority queue. These scale coefficients cover regions rather than individual pixels, so I allocate multiple samples at a time to amortize the overheads introduced in Step 2. For the examples in this chapter, I allocate 64×2^k samples at each iteration, but tuning this parameter offers a trade between speed and quality of adaptation. This also allows us to amortize the costs of ray casting and shading by using SIMD accelerated ray packets [76]. I use the partition traversal algorithm in [52] throughout my system to operate on large groups of (possibly incoherent) rays and shade samples.

The sample locations in the image plane are determined by importance sampling the shape of the scaling function, while samples in other dimensions are chosen via random sampling. The right images in Figure 4.5 show the sample distributions for LeGall 5/3 and Daubechies 9/7 wavelets. The importance sampling requires precomputing a cumulative distribution function (CDF) once for each scale basis function. Note that I only need the CDF for the 1D basis function ϕ, since I importance sample each dimension independently. The plots on the left of Figure 4.5 show the 1D basis functions that I use to compute the CDF for each dimension. After the coefficient is sampled, it is re-inserted in the priority queue, so that it can receive more samples in later iterations if needed.

Step 4—Iterate: If there are more samples in the budget, the algorithm returns to Step 2. There, the priorities $P(S_{k,ij})$ will be updated (in general, the priority for the scale coefficient

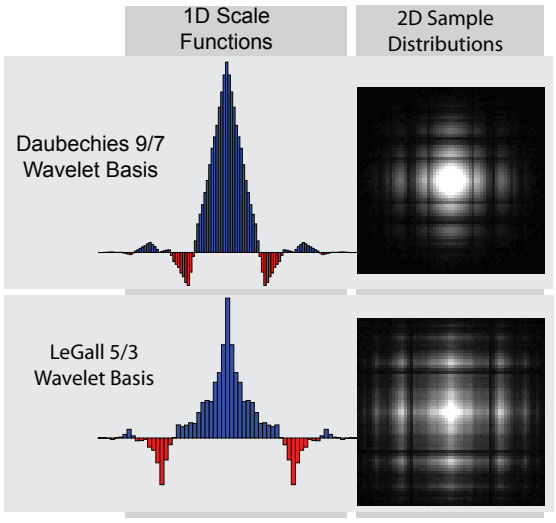


Figure 4.5: The tables used to compute the CDFs for Step 3 and the resulting sample distributions for LeGall 5/3 and Daubechies 9/7 scale functions. On the left, blue values are positive and red are negative. These are scale functions from level $k = 4$.

just chosen will decrease since its variance is reduced—other scale coefficients that overlap its support will also be affected). Then, the algorithm moves to Step 3, sampling the new highest-priority scale function.

In its current design, my algorithm operates with a fixed sample budget, and is finished when this budget is depleted. It would also be possible to use a quality metric as a stopping criteria, such as stopping when there is no measured variance greater than some epsilon. However, since any measure of variance is only approximate, it may be better for the user to visually inspect the results and incrementally add more samples until visual convergence is achieved.

4.4.2 Adaptive Wavelet Reconstruction

At the conclusion of the sampling phase, we have reduced the variance for the coarse scale coefficients in smooth regions of high variance, as well as the fine scale coefficients near edges. The variance that remains is noise, and should be removed by my wavelet reconstruction.

Noise from undersampled Monte Carlo integration appears as fine-grained jump discontinuities in the image, and so should be captured by the fine-scale wavelet coefficients. Therefore, to reduce noise, we can simply reduce the magnitude of the fine-scale wavelet coefficients. One simple approach would be to just ignore coefficients with low magnitudes, as in standard image compression, or threshold the wavelet magnitudes [66]. The sampling stage, however, provides more information on the expected reconstruction error in the form of the variance in each coefficient:

$$\Delta^{\alpha}_{k,ij} = \sqrt{\left\langle \sigma^2(\widetilde{B}), \left(\Psi^{\alpha}_{k,ij}\right)^2 \right\rangle}. \tag{4.10}$$

The square root above is necessary to convert a variance measure to a standard deviation error.

The inner product $\left\langle \sigma^2(\widetilde{B}), \left(\Psi^{\alpha}_{k,ij}\right)^2 \right\rangle$ is computed similarly to $\left\langle \sigma^2(\widetilde{B}), \Phi^2_{k,ij} \right\rangle$ in Step 2 (Equation 4.9) of the adaptive sampling stage, with two exceptions. First, I perform a full DWT using the squares of both high-pass and low-pass filters. Second, the functional variance is computed as the squared difference of the maximum and minimum sample intensities: $\sigma^2(F) = (I_{\max} - I_{\min})^2$. This is used instead of the squared Mitchell contrast in Equation 4.5. While the sampler should be weighted towards darker regions to reduce error there, the reconstruction should smooth both dark and bright regions equally.

The standard deviation in Equation 4.10 provides a range of statistically valid values for the wavelet coefficients. Whereas standard Monte Carlo integration uses the middle of this range, or the average of the samples, I instead take the value of smallest magnitude. This is equivalent to choosing the smoothest image which fits the chosen rendering samples.

Subtracting the standard deviation from the magnitude of the wavelet coefficients gives this result:

$$W^{\alpha}_{k,ij} = \operatorname{sign}(\widetilde{W}^{\alpha}_{k,ij}) \cdot \max\left(0, |\widetilde{W}^{\alpha}_{k,ij}| - c_s \cdot \Delta^{\alpha}_{k,ij}\right), \tag{4.11}$$

where $\widetilde{W}$ are the wavelet coefficients from the pixel means, and c_s (the smoothing constant)

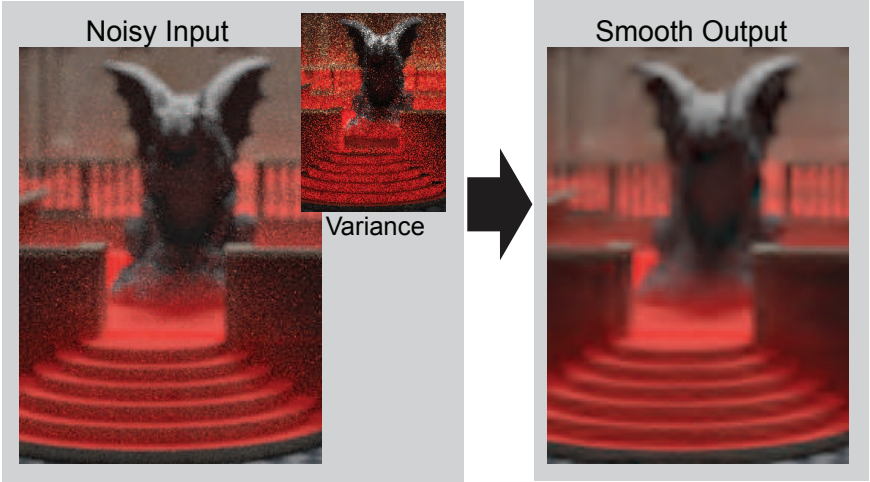


Figure 4.6: Wavelet image reconstruction takes a noisy image and a variance image ($\sigma^2(\widetilde{B})$ in Equation 4.10) as input, and produces a smooth image as output. Here I show close-ups of the out-of-focus gargoyle in Figure 4.1.

is a user-supplied constant that provides a trade-off between noise and wavelet artifacts. Larger values of c_s make smoother images but may introduce ringing around edges or produce a blocky reconstruction. The specific form of the wavelet artifacts depends on the particular wavelet basis used. To avoid coarse scale artifacts, i damp out the smoothing by renormalizing the $\left(\Psi^{\alpha}_{k,ij}\right)^2$ filters to sum to a value < 1. This is analogous to how wavelet compression methods allocate more bits to coarser scale coefficients. For all of the examples in this chapter, i renormalize to $2^{-1/2}$, and i set $c_s = 1$, but it may be possible to tune these values for smoother or sharper results.

As shown in Figure 4.6, my wavelet reconstruction simply requires the noisy results from the adaptive sampling stage and a per-pixel and per-color-channel variance image as input, and successfully removes almost all noise. Note that the entire image reconstruction phase requires at most 2 DWTs (for Equation 4.10 and synthesis from Equation 4.11), and is therefore very efficient.

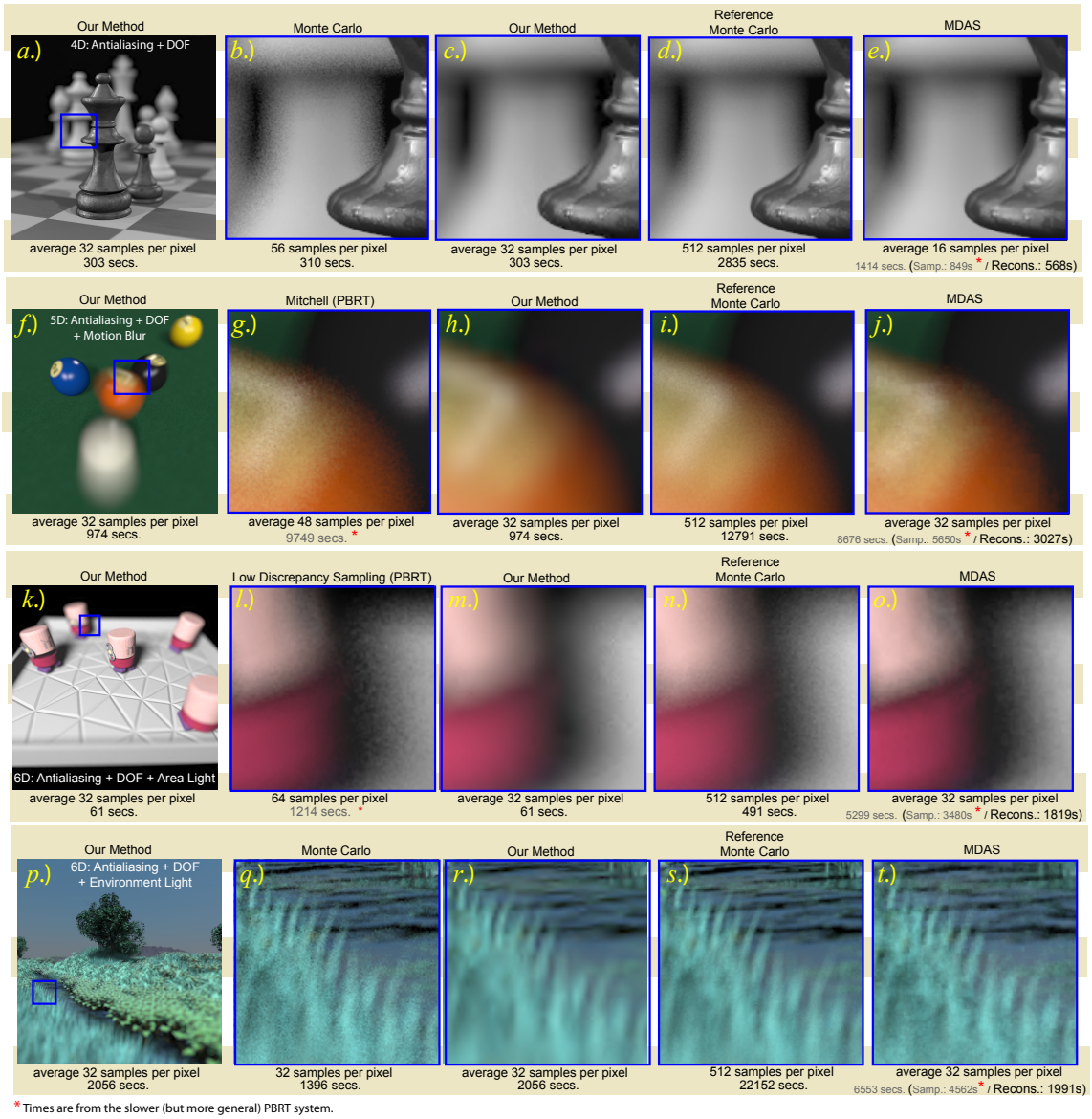


Figure 4.7: All images were rendered at 1024x1024. The problem dimensionality increases from 4D in the top row to 6D in the bottom row. Top row is a 4D scene with antialiasing and depth of field. The middle row is a 5D scene with antialiasing, depth of field, and motion blur, and the bottom row is a 6D scene with antialiasing, depth of field, and area lighting. My method consistently achieves near reference quality images with only 32 samples per pixel for all of these examples. MDAS's results degrade as the problem dimensionality increases, and other methods generate noisy results at low sample counts.

4.5 Results

All results in this chapter were generated on a laptop with a 2.8 GHz Core2 Extreme processor and 3 GB of RAM using one thread, and all images were rendered at 1024×1024.

The five scenes I use are intended to represent a broad range of applications. The "toasters" scene in Figures 4.2 and 4.7 is a simple scene with only 11k triangles, a single rectangular area light, and Phong shading to reduce ray-tracing and shading costs and highlight my system's low overhead. The "chess" scene in the top row of Figure 4.7 has 50k triangles and 9 point lights. The black queen and pawn in the foreground use more complex shaders with bump-mapping, gloss-mapping, mip-mapping, and PBRT's "substrate" material. The "pool" scene in the second row of Figure 4.7 has 57k triangles, 9 point lights, complex shaders, and includes time-varying motion blur effects. The "chess" scene and the "pool" scene (without depth of field) were originally used for the multidimensional adaptive sampling paper [27] using PBRT [54]. The "plants" scene in the third row of Figure 4.7 is from the PBRT distribution and has very high geometric complexity. With over 12k instanced plants and trees, it effectively has over 19 million triangles, many of which are smaller than a pixel. My packet ray tracer is slower on this scene due to the incoherence of the rays relative to the complex geometry. Finally, I added a gargoyle statue to the "sibenik" scene in order to create Figures 4.1*d* and 4.6. This scene has a total of 251k triangles with 1 point light and 1 area light. It uses purely Lambertian materials to demonstrate diffuse interreflections. I added a small red ambient component to the red rug to exaggerate the red color-bleed onto the walls and the gargoyle. I did not include this scene in Figure 4.7 because MDAS is not built to run on this higher dimensional scene. While this chapter is focused on still image rendering, I also include a video of the "chess" scene with animated camera view in the supplementary material which shows the temporal coherence of my method.

4.5.1 Comparisons to Monte Carlo, LDS, Mitchell, MDAS

I have already compared to basic adaptive sampling algorithms in Figure 4.2. In Figure 4.7, I compare my adaptive algorithm to Monte Carlo, low discrepancy sampling (LDS), Mitchell's adaptive sampler [48], and multidimensional adaptive sampling (MDAS) [27].

The implementations I have for these algorithms are in the PBRT system which focuses more on generality than ray casting speed, while I use a speed optimized packet ray tracer for my system and Monte Carlo. Therefore the ray casting times for these systems should not be directly compared to mine. The scenes increase in dimensionality from the top row to the bottom. The chess scene in the top row antialiases the image dimensions and simulates camera depth of field for a 4-dimensional rendering problem. Refer to [27] for comparisons to the Mitchell adaptive sampling algorithm using this scene. The middle row is a 5D problem with antialiasing, depth of field, and motion blur. The scenes in the bottom two rows are 6D with both scenes using antialiasing and depth of field. The "toasters" scene uses a rectangular area light, and the "plants" scene uses environment map lighting.

Out Method: With an average of only 32 samples per pixel (Figure 4.7*a*,*c*,*f*,*h*,*k*,*m*,*p*, and *r*), my method faithfully reproduces the out-of-focus areas that require considerable integration over the camera aperture (and over the time dimension for Figure 4.7*f* and *h*, and over the light source for Figure 4.7*k*,*m*,*p*, and *r*). In these smooth but high variance areas, my adaptive sampling delivers more samples to the coarser scale coefficients. With a low variance estimate at the coarse scale coefficients, the wavelet reconstruction synthesizes a noise-free result.

Monte Carlo: My basic Monte Carlo renderer, without adaptivity, requires at least 512 samples in Figures 4.7*d*, *i*, *n*, and *s* to achieve comparable results. I use 56 samples in Figure 4.7*b* to provide a comparable time comparison, and 32 samples in Figure 4.7*q* to compare at the same sample count. At these low sample densities, Monte Carlo introduces significant noise. For the "chess" scene, my algorithm's overhead is low enough that it is as fast with 32 samples per pixel as Monte Carlo using 56 samples, even though I use SIMD optimized packet ray-tracers.

Low Discrepancy Sampling: LDS generates noisy results in Figure 4.7*l* with 64 samples per pixel. As predicted by Bahvalov's theorems, LDS's benefits over basic Monte Carlo are significantly reduced for this higher dimensional example. Note that I use LDS from the PBRT system which focuses more on generality and accuracy than speed. For the "toasters" scene, I expect a speed optimized version may achieve comparable time to my system at about 64 samples.

Mitchell: Mitchell's adaptive sampler [48] is a popular image-space adaptive technique. It produces noisy results for the motion blurred region in Figure 4.7*g*. Similar to Algorithm 1 in Figure 4.2, it adapts to only fine-scale per-pixel variance, and may therefore sample unevenly in regions that should be smooth. Also, it only uses two passes of adaptivity, and will not iteratively continue to provide samples to this high variance region.

Multidimensional Adaptive Sampling: MDAS is the state-of-the-art adaptive sampling method, and produces a high quality result with only 16 samples per pixel in Figure 4.7*e*. It uses a sophisticated reconstruction algorithm, and therefore requires fewer samples than my method for this example. However, it introduces significant overhead and takes about 4.7× longer than my system at 32 samples per pixel. 849 seconds are spent in MDAS's sampling stage, and 568 seconds in image reconstruction, for a total of 1414 seconds. This is only 2× faster than my Monte Carlo renderer with 512 samples per pixel (2835s vs. 1414s). This disparity is partially due to the fact that I use a speed optimized ray-tracing and shading system, and MDAS uses PBRT, which is optimized for generality rather than speed. However, this only accounts for some of the time spent in MDAS's sampling stage, and my method's full rendering pipeline is almost 2× faster than MDAS's reconstruction stage alone. Thus, my method requires significantly less overhead independent of the particular rendering architecture used.

Moreover, MDAS's results degrade as the problem dimensionality increases. Blocky artifacts begin to appear for the 5D image in Figure 4.7*j*, and are significantly more apparent

in the 6D image in Figure 4.7*o*. It is difficult to use MDAS with more samples per pixel, because MDAS requires 400 bytes per sample (about 12GB for a 1024×1024 image at 32 samples per pixel). My system does not need to store the samples, and so scales better to higher sample densities. I tiled the image to 8×8 in order to render Figure 4.7*o* with 32 samples per pixel. MDAS employs best-candidate sampling for better sample distributions and performs a per-sample k-nearest neighbors search for high quality reconstruction. These approaches work well for 4D scenes like Figure 4.7*e*, but they both run in time exponential in the number of dimensions while their benefit in accuracy decreases exponentially. This explains why MDAS takes 4× longer to render the 6D scene in Figure 4.7*o* at only 2× the number of samples than in Figure 4.7*e*.

4.5.2 Generality, Efficiency, and Visual Consistency

Generality: Due to my algorithm's insensitivity to the problem dimensionality, my algorithm is 5× faster at rendering the 6D scene in Figure 4.7*k* than the 4D scene in Figure 4.7*a* because of the simpler geometry and shaders in the "toasters" scene. To further emphasize the point I removed all but one of the point lights in Figure 4.7*a*, added an area light, and rendered Figure 4.8*a*. With fewer lights overall, it takes less time for my system to render this image than Figure 4.7*a* (211s vs. 303s).

The "plants" scene in the bottom row of Figure 4.7 and in Figure 4.1*c*, and the "sibenik" scene in Figure 4.1*d* present a stress test for my system by introducing large amounts of variance throughout the scene. For the "plants" scene, variance is introduced by environment lighting and sub-pixel geometry. In Figure 4.7*r*, my algorithm captures the out-of-focus wisps of grass, and produces smooth reflections in the water which are noisy even for Monte Carlo with 512 samples per pixel in Figure 4.7*s*. The "sibenik" scene uses Monte Carlo path tracing for one-bounce diffuse interreflections. Note the red color-bleed onto and around the gargoyle statue in the middle as well as the region under the archway on the right that is lit only by indirect illumination.

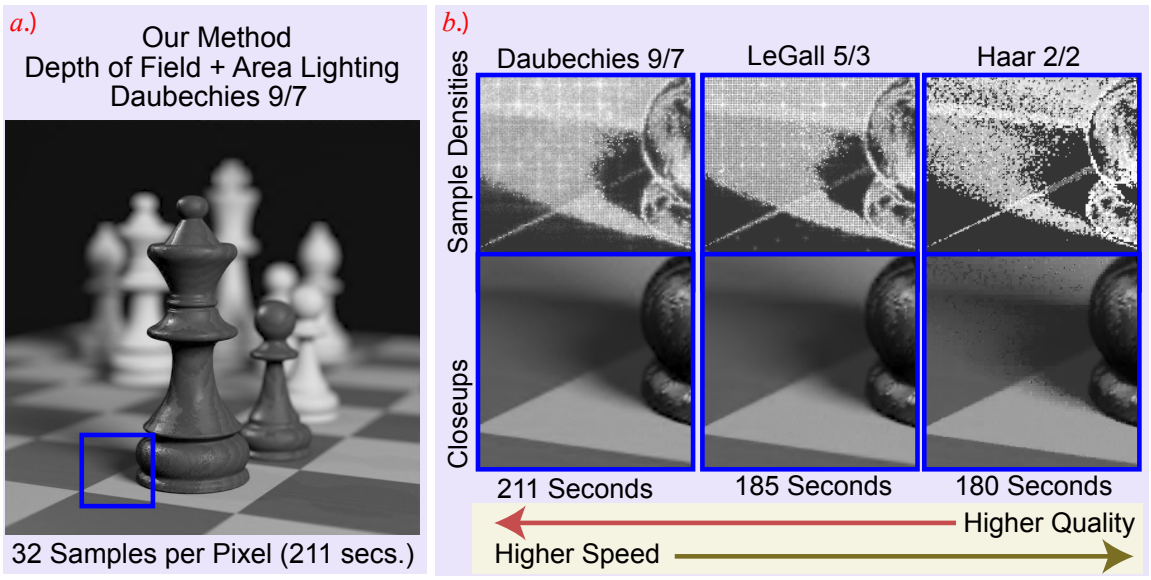


Figure 4.8: A 6D render with depth of field and area lights rendered with an average of 32 samples per pixel. Different wavelet bases offer different speed vs. quality. Haar is the fastest but produces blocky artifacts. LeGall is also fast, and offers reasonable quality. The Daubechies 9/7 filter bank generates the smoothest images.

Variance is high everywhere in the "plants" and "sibenik" scenes, so the adaptive sampler cannot identify local regions to adapt to. Nonetheless, my algorithm still seeks out the regions of smooth variance, and samples these at a lower resolution. Note the multi-scale grid patterns in the sample distributions in the insets of Figure 4.1*c* and *d*. So I still achieve high-quality results with an average of only 32 samples per pixel. Standard Monte Carlo path tracing using my optimized ray tracer requires at least 512 samples per pixel to generate a comparable quality image. This takes over 2 hours for the sibenik scene and over 7 hours for the "plants" scene. At 512 samples, Monte Carlo is still noticeably noisier, but my result with 32 samples has some perceptible wavelet artifacts.

Efficiency: Table 4.1 breaks down the overheads of my algorithm for rendering Figure 4.8 with the Daubechies 9/7 filter bank. Ray-tracing (including the initial sparse sampling and shading) takes 176.5s and my algorithm takes only 34.5s of the total 211s. Thus, my overhead is low, even though I use an optimized packet ray-tracer and shading system. All overhead costs can be adjusted by increasing the amortization (sending more rays for each

Algorithm Component	**Time (Seconds)**
Sampling: Step 1. Sparse RT and Shading	14.5s
Sampling: Step 2. Update Wavelets	12.2s
Sampling: Step 2. Update Variance	10.6s
Sampling: Step 2. Update Priorities	10.7s
Sampling: Step 3. Adaptive RT and Shading	162s
Reconstruction	< 1s
Total	**211 s**

Memory Overheads	**Memory (MBytes)**
Intensity Images for Wavelet Updates (4 float)	16 MB
Intensity Images for Variance Updates (2 float)	8 MB
Image for per-pixel mean (1 RGB)	12 MB
Min/Max Images for per-pixel contrast (2 RGB)	24 MB
Image for per-pixel sample count (1 int)	4 MB
Images for priority queue (2 float, 4 int)	24 MB
Total	**88 MB**

Table 4.1: A breakdown of the timings and memory overheads for generating Figure 4.8 with Daubechies 9/7 wavelets. The time is mostly dominated by ray-tracing and shading costs, and my memory overheads are low.

coefficient in Step 3) or changing the wavelet basis to a simpler basis such as LeGall. The wavelet reconstruction takes less than 1 second, and is dependent on the size of the image alone. I also include all significant memory overheads required by my system. In today's scenes, with 100s of megabytes worth of texture data, 88MB is relatively minor. This size is only dependent on the size of the rendered image so will not change with increased sample densities, scene complexity, or problem dimensionality.

Comparing Wavelet Bases: The close-ups in Figure 4.8*b* compare results using 3 popular wavelet bases. The Haar 2/2, LeGall 5/3, and Daubechies 9/7 filter banks offer different levels of speed and quality. Note in the sample count images in Figures 4.5 and 4.8*b* that the Daubechies filter bank has a more circular and smoother 2D projection as compared to LeGall and Haar. Thus Daubechies provides the smoothest results, but is also slower due to its wide filters. In general, smooth symmetric wavelet bases appear to work best.

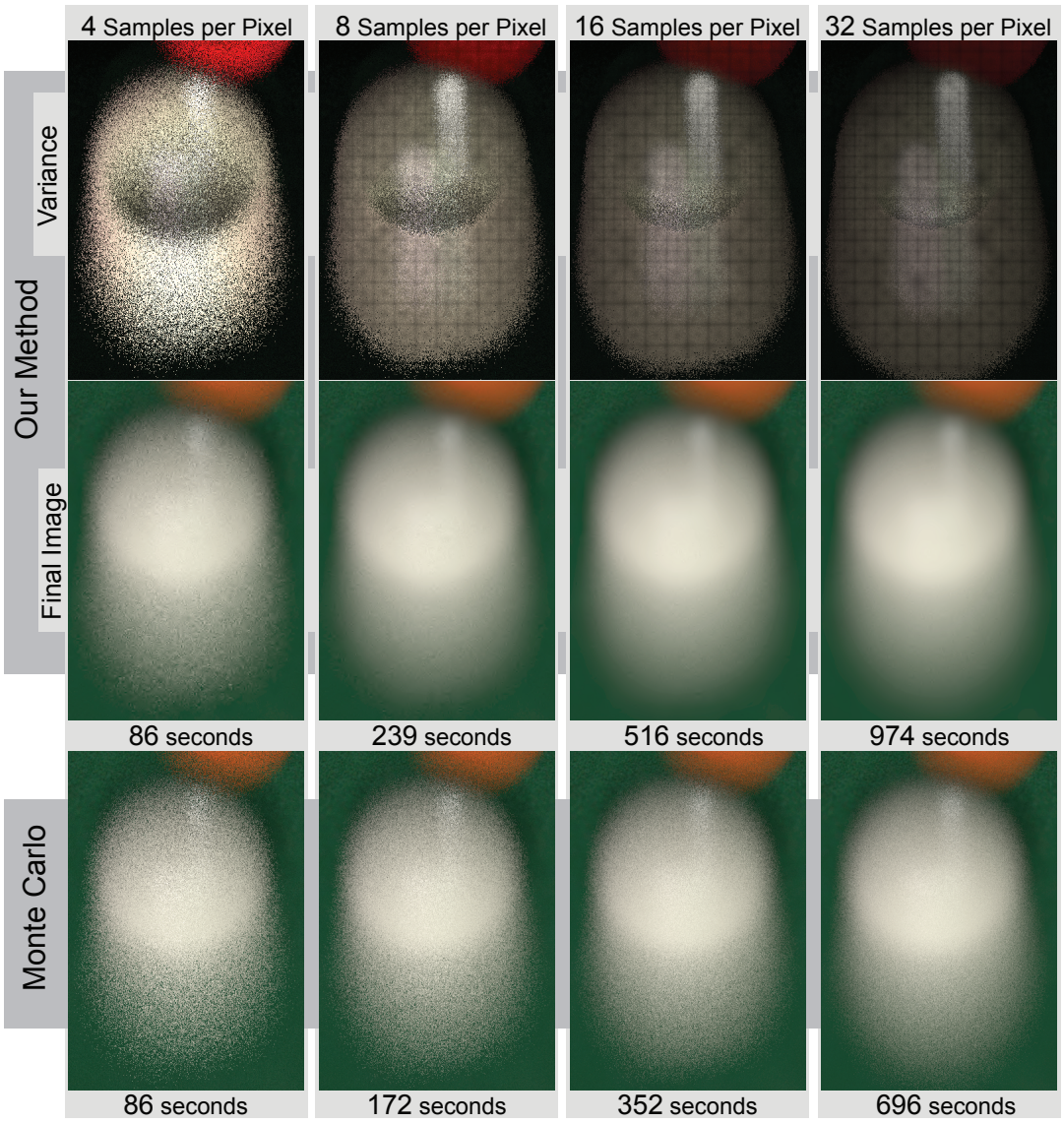


Figure 4.9: Close-ups of a difficult region in Figure 4.7*f*. The cue ball is motion blurred and out-of-focus. At an average of 16 samples, my method produces a mostly smooth result with only minor wavelet artifacts. At 32 samples, my result is essentially converged. Monte Carlo results are noisy by comparison. In the top row, I include scaled up images of variance after the adaptive sampling stage to illustrate the coarse sampling strategy used in smooth regions.

Visual Consistency: Figure 4.9 contains close-ups of the cue ball in Figures 4.1*b* and 4.7*f*, and demonstrates the progression from initial sampling to convergence. After my initial sampling of the image at 4 samples per pixel, I have a high variance estimate, as shown in the variance image in the top left. Even at this low sample count, my result looks quite

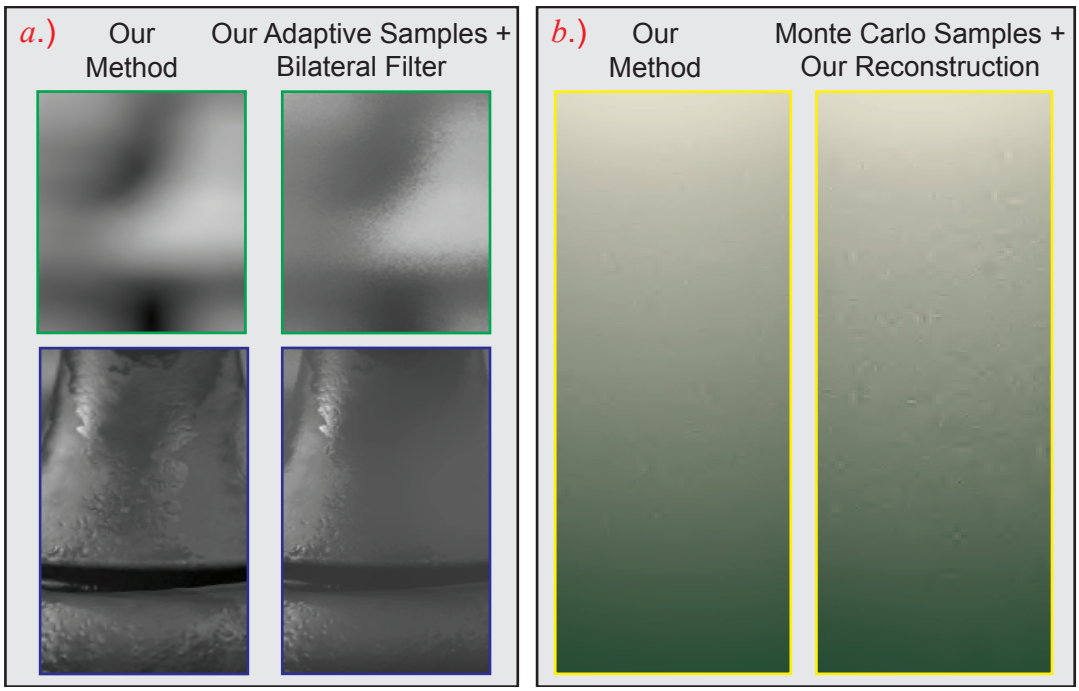


Figure 4.10: My wavelet sampler and reconstruction may be used independently, but are more powerful together. In a.), I use a bilateral filter on the output from my adaptive sampling stage on the "chess" scene. My adaptive samples reduce the variance significantly, but the bilateral filter is unable to remove the noise in the top images, and loses important detail in the bottom images. In b.), I use my wavelet reconstruction on the output from pure Monte Carlo on the "pool" scene. The wavelet reconstruction removes almost all noise, but without my adaptive samples, it introduces some subtle artifacts.

reasonable, with only moderate noise and some wavelet artifacts. At an average of 8 samples per pixel, my algorithm has iteratively distributed samples, and the grid structure of a coarse wavelet level appears in the variance image. At this point, I have a relatively low variance estimate at this coarse level, and so after removing the variance at the finer levels, I achieve a mostly smooth result. At 16 samples per pixel, the grid of a finer wavelet level appears around the structure of the cue ball, and the highlight at the top, and most of the artifacts are gone. Finally, at an average of 32 samples per pixel, my result is essentially converged even though I only have an accurate estimate at the low-resolution wavelet grid. My adaptive sampler leaves most of the pixels with high variance, which is subtracted by my wavelet image reconstruction. Monte Carlo, by comparison, is noisy even with 32 samples.

Sampling vs. Reconstruction It is also possible to use my adaptive sampler and my wavelet reconstruction independently. While each stage has many benefits, in its own right, they work best together. Figure 4.10*a* shows results of my adaptive sampler used with a bilateral filter, and Figure 4.10*b* uses standard Monte Carlo with my wavelet reconstruction. My adaptive sampler significantly reduces the variance of the difficult out-of-focus regions in Figure 4.10*a*, and the bilateral filter can smooth some of the remaining noise. However, some noise remains because the filter cannot take advantage of the fact that my sampler provides more precise results around coarser wavelet nodal points. Also, this bilateral filter does not differentiate between noise from variance and oscillating texture detail which can resemble noise, so some surface detail is washed away.

My variance-based wavelet reconstruction can be a powerful tool when used with non-adaptive Monte Carlo samples. It removes most of the noise in Figure 4.10*b*. However, the coarse wavelet coefficients are not as precise as if they were computed with my adaptive sampler, so some low-frequency noise remains. This low-frequency noise appears as wavelet artifacts in the smooth regions.

4.5.3 Discussion

Quality/Speed Settings: There are several key points of my algorithm that allow tuning of quality and speed parameters. However, we should emphasize that I didn't change any of these settings for the results in this chapter. To reiterate, I use the Daubechies 9/7 filter bank, I set number of rays cast per adaptive iteration to 64×2^k, I set the reconstruction smoothing constant $c_s = 1$, and I renormalize the sampling variance filters to sum to 1.05 and the reconstruction variance filters to sum to $2^{-1/2}$. It may be possible to tune for better results in specific situations.

Limitations: My method uses the wavelet basis to adapt to different sources of variance, including edges and smooth image features, and also to reconstruct a smooth image even

in regions of high variance. When used for image compression, different wavelet bases can exhibit different forms of artifacts, including blockiness and ringing around edges. At low sample densities with too few samples to provide a precise result, I assume a smooth result, and my reconstruction algorithm effectively performs a wavelet compression on the results. For piece-wise smooth natural images, this is generally a reasonable assumption. However, when my assumption is incorrect, some wavelet artifacts may appear. Most of the results in this chapter are essentially converged and so do not exhibit significant artifacts. Figure 4.11 show two examples of wavelet artifacts in the two most difficult scenes, the "plants" scene and the "sibenik" scene. Figure 4.11a is an edge in the middle of a smooth high variance region, and my wavelet reconstruction produces some ringing along this edge. The grass under the shadow of the tree in Figure 4.11b is somewhat overly smoothed. These cases are difficult for Monte Carlo approaches, and require more samples to model accurately. Nonetheless, these subtle artifacts affect only a small region of the image, are largely imperceptible when looking at the full picture, and the benefits with respect to Monte Carlo are shown clearly in Figure 4.7. Moreover, these artifacts are less distracting than the systemic noise introduced by an equally undersampled Monte Carlo simulation in Figure 4.11. It may also be possible to use depth and normal information in an approach similar to the coherence maps used in McCool [46] to get better statistical data for these regions.

4.6 Conclusions and Future Work

I have presented adaptive wavelet rendering, a new method to adapt to both edges and smooth regions of high-dimensional variance. My method is general, and renders complex effects like depth of field, area lighting, motion blur, and global illumination. The technique is simple to implement and efficient, with timings often an order of magnitude faster than previous adaptive algorithms, or optimized Monte Carlo ray-tracers.

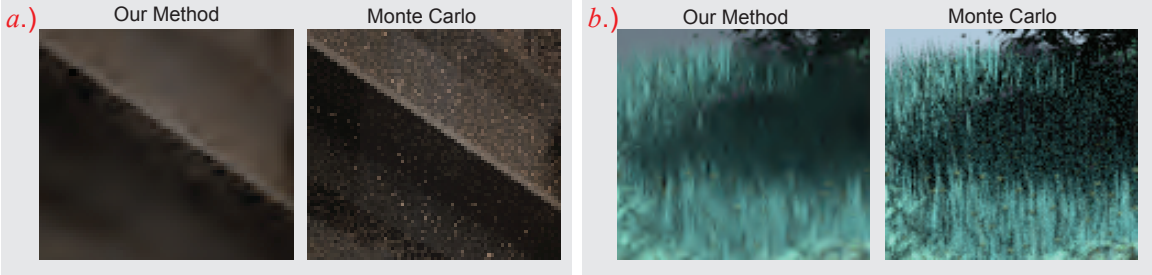


Figure 4.11: Without enough samples (32 samples per pixel in this figure), difficult high variance regions may have artifacts. a.) an edge in the middle of a high variance region. My result has some ringing at the edge. b.) sub-pixel geometry under environment lighting. My method is unable to differentiate variance from visibility and variance from lighting, so my result becomes overly smoothed.

My algorithm currently makes no effort to sample optimally in any dimension other than the image plane. By doing so, I avoid the curse of dimension. However, even scenes with high-dimensional effects are often locally low-dimensional. For example, a stationary object doesn't need motion blur. While my algorithm handles this example well, and will not spend a lot of samples for this event, it may be worthwhile to use an approach that can take better advantage of these locations of low-dimensionality.

In this chapter, I focus on rendering still images, and reserve animated scenes for future work. Preliminary results demonstrate that my method produces smooth animations without temporal artifacts (see video in the supplementary material). Moreover, the time dimension offers extra opportunities for further adaptive sampling and reconstruction. One simple approach would be to expand to a 3D wavelet basis for faster rendering of animated sequences.

I believe my algorithm may be relevant to applications beyond rendering, given the very successful use of wavelets in other domains. It should be emphasized that the only part of my method that may be specific to graphics is the assumption that the image signal is locally smooth. However, many signals studied in other applied sciences have similar characteristics, and my algorithm should be viewed as a general method for high quality parametric integration.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis and related published works [51, 52, 50] offer several methods to extend coherent ray tracing to complex light transport effects.

The beam tracer in Chapter 2 is considerably faster than a one-at-a-time ray tracer for coherent rendering situations. In some cases, where the scene's geometry is large relative to the target sample density, the beam tracer is faster than an optimized packet ray tracer. It also offers qualitative benefits beyond a traditional ray tracer: efficient anti-aliased primary visibility as well as analytic, noise-free soft shadows.

Chapter 3 introduces new algorithms to robustly accelerate the incoherent secondary rays required by a Whitted ray tracer. Specifically, I present *partition traversal*, a new BVH traversal algorithm which is more robust than other packet traversal algorithms to incoherent ray packets. I also introduce a new method to generate tight bounding frusta around general ray packets. I provide a thorough empirical evaluation of these and other coherent ray tracing techniques within a fully interactive Whitted ray tracer.

Finally, in Chapter 4, I propose a new adaptive Monte Carlo sampling and reconstruction algorithm to handle general combinations of high-dimensional effects, such as depth of field,

motion blur, global illumination, and area lighting. The algorithm is specifically designed with coherent ray tracing in mind. It measures variance in a multi-scale wavelet basis, and by adapting to regions of variance, rather than individual pixels, it allows algorithmic amortization using coherent ray tracing. Moreover, the novel wavelet reconstruction makes it possible to sample at an effectively lower resolution in regions of smooth variance, like the blur from a camera lens, and sample more densely near image edges. Adaptive wavelet rendering achieves smooth, near reference quality results with only 32 samples per pixel for many difficult high-dimensional light transport effects.

5.2 Future Work

The diverse nature of these works indicates the broad potential for future work in coherent ray tracing. My work provides a solid base for several directions of future research.

The beam tracer in Chapter 2 proves that beam tracing is a viable option when there are many highly coherent ray queries. This work focuses on primary visibility, point-light shadows, and soft shadows from area light sources, but there are many other effects which are prone to aliasing and noise. For example, beam tracing may be used for noise-free depth of field, by tracing a camera aperture shaped beam for each image pixel. Similar to the algorithm for soft shadows, the blurry image pixels could be computed analytically. Also, in order for beam tracing to reach more mainstream appeal, it needs to be optimized for more finely tessellated geometry.

The Whitted ray tracer in Chapter 3 will benefit from the increased instruction-level and data-level parallelism available on future hardwares. However, many attributes change with each new hardware generation, from the structure of memory hierarchies to the instruction set architecture. Already, SSE is being replaced by AVX, and some future highly parallel architectures, such as Intel's Larrabee, have much smaller memory caches than the standard general purpose CPU. Ray packet algorithms will need to change with the hardware, or (as

ray packet algorithms reach broader adoption) the hardware will have to change with the algorithms. Also, thorough empirical evaluation of coherent ray tracing should be extended to include more rendering algorithms, such as distributed ray tracing, path tracing, and photon mapping. Perhaps it is time for a new standardized test suite, similar in spirit to BART [42], to evaluate ray tracing algorithms under various stresses in a verifiable way.

My research into adaptive wavelet rendering in Chapter 4 demonstrates the power of rendering directly to a wavelet basis, and suggests multiple avenues for future research. In my work, only the image-space dimensions are considered for adaptation and noise removal. It would be interesting to extend to a 3D basis and incorporate time, since there should be significant coherence in that dimension as well. Also, consider that the wavelet basis is a powerful tool for image and video compression. By rendering directly to a compressed basis, it may be possible to perform remote interactive rendering. High quality images could be simultaneously computed and compressed on a server (or cluster of servers) and transmitted via broadband to a mobile client.

In general, now is an exciting time for coherent ray tracing research, and the future appears bright. The algorithms have matured and are ready for inclusion into commercial systemsm, and industries are betting heavily on coherent ray tracing. Both Intel and NVidia are tuning coherent ray tracing algorithms to run on their newer hardwares [59, 2]. Production rendering companies, like Pixar and Dreamworks Animation, are relying more on ray tracing and are looking to ray coherence to make complex rendering tractable [12, 7]. With the increased commercial appeal, the need for new research should grow as well.

My work and the recent interest from industry suggests that the impact of coherent ray tracing should be a higher priority consideration in all future rendering research. There is a wealth of algorithms that were designed specifically with a one-at-a-time ray tracer in mind. There are few that directly map to coherent ray tracing, and there are few more that have been adjusted to be more amenable to coherent ray tracing. In order to get the most from coherent ray tracing algorithms, they should be incorporated from the start of algorithm

design.

It would seem that truly interactive photorealistic rendering may be just around the corner. The algorithms presented in this thesis help bring us closer to this goal. With the rapid pace of coherent ray tracing research and the promise of future hardwares with dizzying amounts parallelism, it seems possible that we will soon be visiting and interacting with virtual worlds that are indistinguishable from reality.

Chapter 6

Bibliography

[1] Maneesh Agrawala, Ravi Ramamoorthi, Alan Heirich, and Laurent Moll. Efficient image-based methods for rendering soft shadows. In *ACM SIGGRAPH 00*, pages 375–384, 2000.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of High-Performance Graphics 2009*, 2009.

[3] John Amanatides. Ray tracing with cones. In *ACM SIGGRAPH 84*, pages 129–135, 1984.

[4] James Arvo. Applications of irradiance tensors to the simulation of non-Lambertian phenomena. In *ACM SIGGRAPH 95*, pages 335–342, 1995.

[5] Nikolai S. Bahvalov. On approximate calculation of multiple integrals. Technical report, Vestnik Moscow Univ., 1959.

[6] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. *ACM TOG (SIGGRAPH 03)*, 22(3):631–640, 2003.

[7] Carsten Benthin, Solomon Boulos, J Dylan Lacewell, and Ingo Wald. Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces. Technical Report UUSCI-2007-011, 2007.

[8] Mark R. Bolin and Gary W. Meyer. A perceptually based adaptive sampling algorithm. In *ACM SIGGRAPH 98*, pages 299–310, 1998.

[9] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface*, pages 177–184, 2007.

[10] Solomon Boulos, Ingo Wald, and Carsten Benthin. Adaptive ray packet reordering. In *IEEE/EG Symposium on Interactive Ray Tracing*, pages 131–138, 2008.

[11] Solomon Boulos, Ingo Wald, and Peter Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Technical Report UUCS-06-010, 2006.

[12] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. Ray tracing for the movie ćars.ˊ In *IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006.

[13] Petrik Clarberg, Wojciech Jarosz, Tomas Akenine-Möller, and Henrik Wann Jensen. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM TOG (SIGGRAPH 05)*, 24(3):1166–1175, 2005.

[14] A. Cohen, Ingrid Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45(5):485–560, 1992.

[15] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *ACM SIGGRAPH 84*, pages 137–145, 1984.

[16] Franklin C. Crow. Shadow algorithms for computer graphics. In *ACM SIGGRAPH 77*, pages 242–248, 1977.

[17] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In *ACM SIGGRAPH 94*, pages 223–230, 1994.

[18] Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François Sillion. A frequency analysis of light transport. *ACM TOG (SIGGRAPH 05)*, 24(3):1115–1126, 2005.

[19] Kevin Egan, Yu-Ting Tseng, Nicolas Holzschuch, Frédo Durand, and Ravi Ramamoorthi. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM TOG (SIGGRAPH 09)*, 28(3):93, 2009.

[20] Thomas Funkhouser, Ingrid Carlbom, Gary Elko, Gopal Pingali, Mohan Sondhi, and Jim West. A beam tracing approach to acoustic modeling for interactive virtual environments. In *ACM SIGGRAPH 98*, pages 21–32, 1998.

[21] Thomas A. Funkhouser, Patrick Min, and Ingrid Carlbom. Real-time acoustic modeling for distributed virtual environments. In *ACM SIGGRAPH 99*, pages 365–374, 1999.

[22] Djamchid Ghazanfarpour and Jean-Marc Hasenfratz. A beam tracing method with precise antialiasing for polyhedral scenes. *Computers and Graphics*, 22(1):103–115, 1998.

[23] Steven J. Gortler, Peter Schröder, Michael F. Cohen, and Pat Hanrahan. Wavelet radiosity. In *ACM SIGGRAPH 93*, pages 221–230, 1993.

[24] Christiaan P. Gribble and Karthik Ramani. Coherent ray tracing via stream filtering. In *IEEE/EG Symposium on Interactive Ray Tracing*, pages 59–66, 2008.

[25] Baining Guo. Progressive radiance evaluation using directional coherence maps. In *ACM SIGGRAPH 98*, pages 255–266, 1998.

[26] Seymour Haber. Stochastic quadrature formulas. *Mathematics of Computation*, 23(108):751–764, 1969.

[27] Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker, and Henrik Wann Jensen. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM TOG (SIGGRAPH 08)*, 27(3):1–10, 2008.

[28] Eric Haines. A proposal for standard graphics environments. *IEEE Computer Graphics & Applications*, 7(11):3–5, 1987.

[29] David Hart, Philip Dutre, and Donald P. Greenberg. Direct illumination with lazy visibility evaluation. In *ACM SIGGRAPH 99*, pages 147–154, 1999.

[30] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadow algorithms. *Computer Graphics Forum*, 22(4):753–774, Dec. 2003. State-of-the-Art Reviews.

[31] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[32] Vlastimil Havran and Jiří Bittner. LCTS: Ray shooting using longest common traversal sequences. *Computer Graphics Forum (Eurographics 00)*, 19(3):59–70, 2000.

[33] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *ACM SIGGRAPH 84*, pages 119–127, 1984.

[34] Stefan Heinrich and Eugène Sindambiwe. Monte carlo complexity of parametric integration. *Journal of Complexity*, 15(3):317–341, 1999.

[35] Warren Hunt, William R. Mark, and Don Fussell. Fast and lazy build of acceleration structures from scene hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 47–54, 2007.

[36] Warren Hunt, William R. Mark, and Gordon Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 81–88, 2006.

[37] Homan Igehy. Tracing ray differentials. In *ACM SIGGRAPH 99*, pages 179–186, 1999.

[38] James T. Kajiya. The rendering equation. In *ACM SIGGRAPH 86*, pages 143–150, 1986.

[39] Alexander Keller. Hierarchical monte carlo image synthesis. *Mathematics and Computers in Simulation*, 55(1–3):79–92, 2001.

[40] Samuli Laine, Timo Aila, Ulf Assarsson, Jaakko Lehtinen, and Tomas Akenine-Möller. Soft shadow volumes for ray tracing. *ACM TOG (SIGGRAPH 05)*, 24(3):1156–1165, 2005.

[41] Jaakko Lehtinen, Samuli Laine, and Timo Aila. An improved physically-based soft shadow volume algorithm. *Computer Graphics Forum*, 25(3):303–312, 2006.

[42] Jonas Lext, Ulf Assarsson, and Tomas Möller. Bart: A benchmark for animated ray tracing, 2000.

[43] Baoquan Liu, Li-Yi Wei, Xu Yang, Ying-Qing Xu, and Baining Guo. Nonlinear beam tracing on a gpu. Technical Report MSR-TR-2007-34, Microsoft Research, 2007.

[44] Stéphane Mallat. *A Wavelet Tour of Signal Processing, Second Edition (Wavelet Analysis & Its Applications)*. Academic Press, 1999.

[45] Erik Mansson, Jacob Munkberg, and Tomas Akenine-Möller. Deep coherent ray tracing. In *IEEE Symposium on Interactive Ray Tracing*, pages 79–85, 2007.

[46] Michael D. McCool. Anisotropic diffusion for Monte Carlo noise reduction. *ACM TOG*, 18(2):171–194, 1999.

[47] Mark Meyer and John Anderson. Statistical acceleration for animated global illumination. *ACM TOG (SIGGRAPH 06)*, 25(3):1075–1080, 2006.

[48] Don P. Mitchell. Generating antialiased images at low sampling densities. In *ACM SIGGRAPH 87*, pages 65–72, 1987.

[49] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. *ACM TOG (SIGGRAPH 03)*, 22(3):376–381, 2003.

[50] Ryan Overbeck, Craig Donner, and Ravi Ramamoorthi. Adaptive wavelet rendering. *ACM TOG (SIGGRAPH Asia 09)*, 2009.

[51] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. A real-time beam tracer with application to exact soft shadows. In *Eurographics Symposium on Rendering*, 2007.

[52] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large ray packets for real-time Whitted ray tracing. In *IEEE/EG Symposium on Interactive Ray Tracing*, pages 41–48, 2008.

[53] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, 1990.

[54] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 2004.

[55] Zhong Ren, Rui Wang, John Snyder, Kun Zhou, Xinguo Liu, Bo Sun, Peter-Pike Sloan, Hujun Bao, Qunsheng Peng, and Baining Guo. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. *ACM TOG (SIGGRAPH 06)*, pages 977–986, 2006.

[56] Alexander Reshetov. Faster ray packets - triangle intersection through vertex culling. In *IEEE Symposium on Interactive Ray Tracing*, pages 105–112, 2007.

[57] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM TOG (SIGGRAPH 05)*, 24(3):1176–1185, 2005.

[58] Holly E. Rushmeier and Gregory J. Ward. Energy preserving non-linear filters. In *ACM SIGGRAPH 94*, pages 131–138, 1994.

[59] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM TOG (SIGGRAPH 08)*, 27(3):1–15, 2008.

[60] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. Monte carlo techniques for direct lighting calculations. *ACM TOG*, 15(1):1–36, 1996.

[61] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine*, 18(5):36–58, 2001.

[62] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM TOG (SIGGRAPH 02)*, 21(3):527–536, 2002.

[63] Cyril Soler and Francois Sillion. Fast calculation of soft shadow textures using convolution. In *ACM SIGGRAPH 98*, pages 321–332, 1998.

[64] Cyril Soler, Kartic Subr, Frédo Durand, Nicolas Holzschuch, and François Sillion. Fourier depth of field. *ACM TOG*, 28(2):18, 2009.

[65] A. James Stewart and Sherif Ghali. Fast computation of shadow boundaries using spatial coherence and backprojections. In *ACM SIGGRAPH 94*, pages 231–238, 1994.

[66] Gilbert Strang and Truong Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1997.

[67] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[68] Seth Teller and John Alex. Frustum casting for progressive, interactive rendering. Technical Report MIT/LCS/TR-740, 1998.

[69] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *ICCV 98*, pages 839–846, 1998.

[70] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *ACM SIGGRAPH 97*, pages 65–76, 1997.

[71] Ingo Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[72] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM TOG*, 26(1), 2007.

[73] Ingo Wald, Christiaan P Gribble, Solomon Boulos, and Andrew Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, 2007.

[74] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM TOG (SIGGRAPH 06)*, pages 485–493, 2006.

[75] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007.

[76] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Eurographics 01)*, 20(3):153–164, 2001.

[77] Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. Multidimensional lightcuts. *ACM TOG (SIGGRAPH 06)*, pages 1081–1088, 2006.

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

[79] Ruifeng Xu and Sumanta N. Pattanaik. A novel Monte Carlo noise reduction operator. *IEEE Computer Graphics and Applications*, 25(2):31–35, 2005.

[80] Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha. Ray tracing dynamic scenes using selective restructuring. In *Eurographics Symposium on Rendering*, 2007.

Appendix A

Beam Tracing Algorithm Details

This appendix is complementary to Section 2.2. The following sections provide low-level details and pseudocode for beam–triangle intersection and kd-tree traversal, the two key components of our beam tracing algorithm.

A.1 Beam Representation

A 3D ray is defined by an origin, a direction, and a maximum distance along the ray (the hit distance once a hit is found for a primary ray). We represent a beam as four corner rays. In our work, we exploit the SSE intrinsics, a library of macros, for ease of implementation. The atomic datatype used by these macros is the __m128 structure, which packs 4 32–bit single precision floating point values together into one variable. Most standard floating point operations can be performed on all four variables in parallel. In the following pseudocode examples we will replace these intrinsics with more intuitively meaningful terminology (__m128 ≡ float4) and operators.

Instead of representing a beam as an array of four rays (Array of Structures or AOS), it is more efficient to interleave the rays' members into Structures of Arrays (SOA):

```
// SOA representation of 4 3D vectors
struct soavec3 {
  float4 x,y,z;
};

// A 3D Beam is defined by 4 corner rays which meet at a point.
struct Beam {
  soavec3 Origin;   // Beam origin
  soavec3 Dirs;     // Beam corner's directions
  soavec3 InvDirs;  // Inverse of directions
  float4  MinDist;  // Min distance along corner rays
  float4  MaxDist;  // Max distance along corner rays
  int     Signs[3]; // Signs of Dir
  int     pad;      // Keep the structure aligned
};
```

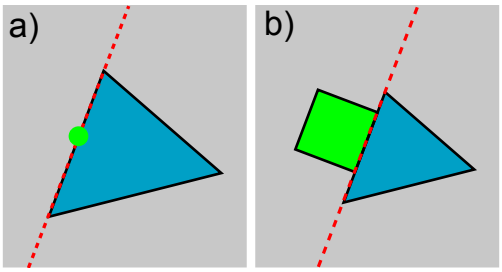


Figure A.1: A difficult intersection test case requiring fuzzy logic. Does the infinitesimally small ray in (a) (green) lie inside or outside the triangle (blue) as determined by its left edge (red)? Clearly the beam in (b) (green) misses the triangle, but floating point accuracy may tell us otherwise.

A.2 Triangle Intersection

In the image plane, the beam's 3 or 4 corner points can all lie on one side of a triangle edge or be split in two. Using SSE2 instructions, we can simultaneously test the 4 beam points against the triangle edge, and store the result in a 4-bit mask:

```
soavec2 diff   = BeamPoints2D - triPoint2D;
float4  dotval = dot( diff,triPerp2D );
bool4   mask   = (dotval < 0);
```

This mask tells us on which side of the plane each of the 4 points resides. If all 4 bits are zeros, all points lie on the outside of the plane, and if all are ones, they lie on the inside. Any other result indicates that the plane splits the 4 points, and more than that, exactly which edges the plane splits. There will be either 2 or 0 intersected edges, and finding the actual intersection points requires two line–plane intersection tests.

We use a switch statement on the mask from the plane test, to split based on the 15 different possible orientations of the points relative to the edge (in reality there are only 11 possible orientations as 0 and 15 are the no split cases and 5 and 10 aren't possible as long as the beam's cross section is convex). We then shuffle the beam points and edges such that we can find the two intersection points in parallel. Computationally, finding these two intersection points is about as expensive as generating one new ray in a conventional ray tracer. However, the branching and the shuffling do add some overhead. Once we have determined the intersection points, it is a simple matter to shuffle the beam's members with the intersection points to generate the new beams.

Robust Splitting With Fuzzy Intersection Tests: What happens when a plane passes through beam corner points as in Fig. A.1b? If we were dealing with rays (Fig. A.1a), this is a difficult question to answer, but when dealing with beams (Fig. A.1b), the other corner points should determine the result. If we're not careful, numerical imprecision could lead us to split this beam when no split is really necessary. Corner points which are within some distance epsilon from the plane shouldn't affect the determination of which side the beam is on and whether to split. These points can be masked out using a fuzzy mask:

```
bool4 fuzzyMask = (Epsilon < abs(dotval));
```

This assures that only the points which can definitively decide the result come into play. Similar fuzzy logic is used throughout our beam tracer.

A.3 KD-Tree Traversal

Below we include pseudocode for our beam–kd-tree traversal algorithm. Note that it is quite similar to the standard ray–kd-tree traversal algorithm. In the actual implementation, there is also a test to see if the plane passes through the origin, and we must visit the left or right child node determined by the signs of the beam's directions.

When the beam must visit both child nodes, we must push a `KDStackNode` onto the `KDStack` to be retrieved later by `GetNextBeam()`. Each `KDStackNode` must hold a pointer to the far `KDNode`, the bounding box of the far node, and the current size of the `MissStack`. All new beams pushed onto the `MissStack` during triangle intersection must continue down the kd-tree as the current beam would. `GetNextBeam()`(not shown here) must determine the next `KDNode` to be visited as well as the next beam for that node and the new `TMin` and `TMax` values for that beam.

```
void Traverse( Beam * CurBeam, KDTree & Tree ) {
  const int mod[] = {1,2,0,1};
  // Current bounding box relative to beam's origin
  AABBox CurBox = KDTree.Box - CurBeam->Origin;
  soavec3 TMin; // Entry distances for each beam ray and axis
  soavec3 TMax; // Exit distances for each beam ray and axis
  // Check if the CurBeam hits the scene's bounding box,
  // and initialize TMin and TMax.
  if ( !BeamVsAABBox(CurBeam,CurBox,TMin,TMax) )  return;
  KDNode* CurKDNode = KDTree.Root;
  while( CurBeam ) {
    while( !CurKDNode->IsLeaf() ) {
      int Axis         = CurKDNode->GetAxis();
      float4 Split     = CurKDNode->GetSplit();
      float4 SplitSub0 = Split - CurBeam->Origin;
      float4 Dist      = CurBeam->InvDirs[axis] * SplitSub0;
      bool4 ltZeroMask = (Dist < 0);
      if ( ! ~ltZeroMask ) {
        // Plane passes behind Origin
        CurKDNode = GetNearChild(CurKDNode,CurBeam);
      } else if(!(Dist < TMax[mod[Axis]]) || !(Dist < TMax[mod[Axis+1]])) {
        // Traverse near node only
        CurKDNode = GetNearChild(CurKDNode,CurBeam);
      } else if(!(TMin[mod[Axis]] < Dist) || !(TMin[mod[Axis+1]] < Dist)) {
        // Traverse far node only
        CurKDNode = GetFarChild(CurKDNode,CurBeam);
      } else {
        // Traverse both nodes
        int     which   = CurBeam->Signs[Axis];
```

```
        AABBox  FarBox  = CurBox;
        KDNode* FarNode = GetFarChild(CurKDNode,CurBeam);
        // Save the far kd-node for later.
        FarBox[which][Axis] = SplitSubO;
        KDStack.push( KDStackNode( FarNode, FarBox, MissStack.size() ));
        // Traverse the near node.
        CurBox[1-which][Axis] = SplitSubO;
        TMax[Axis]            = Dist;
        CurKDNode             = GetNearChild(CurKDNode,CurBeam);
      }
    }
    // Intersect with the triangles in this leaf node.
    int NumTris = CurKDNode->GetNumTris();
    if ( NumTris )
      BeamVsTriList(CurBeam,CurKDNode->GetTriListPtr(),NumTris);
    // Get the next beam and KDNode from the stack
    (CurBeam, CurKDNode) = GetNextBeam(KDStack,CurBeam,TMin,TMax);
  }
}
```

Appendix B

Real-time Whitted Ray Tracing Pseudocode

This appendix provides pseudocode for the Ranged traversal and Partition traversal algorithms in Chapter 3. Pseudocode for Masked traversal can be found in Subsection 3.2.1.

B.1 Ranged Traversal

```
 1: // Traverse a Ray packet, R, through theBVH using Ranged Traversal
 2: void rangedTraverseBVH( Rays R, Frustum F, BVH theBVH )
 3:     BVHCell curCell = theBVH.root;
 4:     Stack<StackNode> traversalStack;
 5:     Index i_a = 0;
 6:     while ( true )
 7:       i_a = getFirstHit( R, F, curCell.AABB(), i_a );
 8:       if ( i_a < size(R) )
 9:         if ( isInner( curCell ))
10:           StackNode node;
11:           node.cell = curCell.farChild( R );
12:           node.i_a = i_a;
13:           traversalStack.pushBack( node );
14:           curCell = curCell.nearChild( R );
15:           continue ;
16:         else  // isLeaf( curCell ) == true
17:           Index i_e = getLastHit( R, curCell.AABB(), i_a );
18:           Triangles T = curCell.triangles();
19:           for ( Index j = 0; j < size( T ); ++j )
20:             if ( frustumIntersectsTriangle( F, T[j] ))
21:               for ( Index i = i_a; i < i_e ; ++i )
22:                 rayIntersectTriangle( R[i], T[j] );
23:       // END if ( i_a < size(R) )
24:       if ( traversalStack.empty() )
```

```
25:          break ;
26:       StackNode node = traversalStack.pop();
27:       curCell = node.cell;
28:       i_a = node.i_a;
29:    // END while ( true )...
30:// END void traverseBVH(...
```

B.2 Partition Traversal

```
 1:  // Traverse a Ray packet, R, through theBVH using Partition Traversal
 2: void partitionTraverseBVH( Rays R, Frustum F, BVH theBVH )
 3:    BVHCell curCell = theBVH.root;
 4:    Stack<StackNode> traversalStack;
 5:    Index I[size( R )];
 6:    for ( Index i = 0; i < size(R); ++i ) I[i] = i;
 7:    Index i_a = 0;
 8:    while ( true )
 9:       i_a = partRays( R, F, curCell.AABB(), I, i_a );
10:       if ( i_a > 0 )
11:          if ( isInner( curCell ))
12:             StackNode node;
13:             node.cell = curCell.farChild( R );
14:             node.i_a = i_a;
15:             traversalStack.pushBack( node );
16:             curCell = curCell.nearChild( R );
17:             continue ;
18:          else  // isLeaf( curCell ) == true
19:             Triangles T = curCell.triangles();
20:             for ( Index j = 0; j < size( T ); ++j )
21:                if ( frustumIntersectsTriangle( F, T[j] ))
22:                   for ( Index i = 0; i < i_a ; ++i )
23:                      rayIntersectTriangle( R[I[i]], T[j] );
24:       // END if ( i_a > 0 )
25:       if ( traversalStack.empty() )
26:          break ;
27:       StackNode node = traversalStack.pop();
28:       curCell = node.cell;
29:       i_a = node.i_a;
30:    // END while ( true )...
31:// END void traverseBVH(...
```

Appendix C

Adaptive Wavelet Rendering Companion

This appendix serves as a companion to Chapter 4. Section C.1 details the Daubechies 9/7 and LeGall 5/3 wavelet filter banks which I found to work best for adaptive wavelet rendering. Section C.2 derives Equation 4.9 which is used to compute the variance at the wavelet basis' scale and wavelet coefficients.

C.1 Daubechies 9/7 and LeGall 5/3 Filter Banks

We found the Daubechies 9/7 and LeGall 5/3 wavelets to work better for our algorithm than the simpler Haar wavelet basis or various other filter banks that we tested. They are both symmetric biorthogonal filter banks. The quality of a wavelet is often measured by the number of vanishing moments: the lowest degree polynomial for which the results of applying the high-pass filter are non-zero. The Daubechies 9/7 filter bank has 4 vanishing moments, so can exactly fit polynomials of cubic degree or lower. LeGall 5/3 has 2 vanishing moments. These are the highest theoretically possible for their support widths. The Daubechies 9/7 wavelet is used for high quality lossy encoding in JPEG 2000 because it is very smooth. LeGall 5/3 is an instance of a binlet, a filter bank that can be implemented as all integer values, so it is used for efficient lossless encoding in JPEG 2000.

Since they are both symmetric, we list only half of the coefficients. The rest can be obtained by reflecting about the first coefficient.

Daubechies 9/7
Analysis Low-Pass:
$(\sqrt{2})$ ×{0.602949, 0.266864, -0.078223, -0.016864, 0.026749}
Analysis High-Pass:
$(1/\sqrt{2})$ ×{1.115087, -0.591272, -0.057544, 0.091272}
Synthesis Low-Pass:
$(1/\sqrt{2})$ ×{1.115087, 0.591272, -0.057544, -0.091272}
Synthesis High-Pass:
$(\sqrt{2})$ ×{0.602949, -0.266864, -0.078223, 0.016864, 0.026749}

LeGall 5/3

Analysis Low-Pass: $(\sqrt{2}) \times(1/8)\times\{6, 2, -1\}$
Analysis High-Pass: $(1/\sqrt{2}) \times(1/2)\times\{-2, 1\}$
Synthesis Low-Pass: $(1/\sqrt{2}) \times(1/2)\times\{2, 1\}$
Synthesis High-Pass: $(\sqrt{2}) \times(1/8)\times\{-6, 2, 1\}$

C.2 Derivation of Equation 4.9

Equation 4.9 accumulates the approximate variance at the scale coefficients from an estimate of variance at the pixels. Equation 4.4 states that the scale coefficients are equal to the inner product of the pixel values with the scale function. For a discrete wavelet basis, this inner product is a weighted sum:

$$S = \left\langle \widetilde{B}, \Phi \right\rangle = \sum_i \widetilde{B}_i \Phi_i, \tag{C.1}$$

where the Φ_i are the discrete scale function's filter coefficients, and the $\widetilde{B}_i$ are the pixel means. Note that we have dropped the wavelet translation and dilation subscripts from S and Φ for simplicity. We seek the variance of the scale coefficient: $\sigma^2(S) = \sigma^2\left(\sum_i \widetilde{B}_i \Phi_i\right)$. This can be computed using the identity:

$$\sigma^2\left(\sum_i c_i x_i\right) = \sum_i c_i^2 \sigma^2(x_i) + \sum_i \sum_{j>i} 2 c_i c_j \mathrm{cov}(x_i, x_j), \tag{C.2}$$

which holds for any set of constants c_i and random variables x_i. If the x_i are uncorrelated random variables, then the covariance term tends to zero, and we have:

$$\sigma^2\left(\sum_i c_i x_i\right) \approx \sum_i c_i^2 \sigma^2(x_i). \tag{C.3}$$

Finally, this gives us:

$$\sigma^2\left(\sum_i \widetilde{B}_i \Phi_i\right) \approx \sum_i \Phi_i^2 \sigma^2\left(\widetilde{B}_i\right). \tag{C.4}$$

The right side of Equation C.4 is equivalent to the right side of Equation 4.9.