Eurographics Symposium on Rendering (2007)
Jan Kautz and Sumanta Pattanaik (Editors)

A Real-time Beam Tracer with Application to Exact Soft Shadows

Ryan Overbeck[1], Ravi Ramamoorthi[1] and William R. Mark[2]

[1]Columbia University
[2]University of Texas at Austin

Abstract

Efficiently calculating accurate soft shadows cast by area light sources remains a difficult problem. Ray tracing based approaches are subject to noise or banding, and most other accurate methods either scale poorly with scene geometry or place restrictions on geometry and/or light source size and shape. Beam tracing is one solution which has historically been considered too slow and complicated for most practical rendering applications. Beam tracing's performance has been hindered by complex geometry intersection tests, and a lack of good acceleration structures with efficient algorithms to traverse them. We introduce fast new algorithms for beam tracing, specifically for beam–triangle intersection and beam–kd-tree traversal. The result is a beam tracer capable of calculating precise primary visibility and point light shadows in real-time. Moreover, beam tracing provides full area elements instead of point samples, which allows us to maintain coherence through to secondary effects and utilize the GPU for high quality antialiasing and shading with minimal extra cost. More importantly, our analysis shows that beam tracing is particularly well suited to soft shadows from area lights, and we generate essentially exact noise-free soft shadows for complex scenes in seconds rather than minutes or hours.

1. Introduction

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 [SWZ96].

Ray tracing has a long history in graphics [Whi80], 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 [WSBW01], frustum proxies for kd-tree traversal [RSH05, WIK*06, TA98, HB00], and make efficient use of the register SIMD (e.g., SSE) instructions available on most modern processors [WSBW01, RSH05, WIK*06]. 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 [RSH05]). Furthermore, initial indications are that the benefits of these methods are relatively modest for secondary effects such as soft shadows. The recent results from [BEL*07] 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 [HH84] 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 several reasons. The basic geometry intersection tests become significantly more complicated when moving from rays to beams. Moreover, there has not been significant success in applying acceleration structures to beam tracing.

In this paper, we develop novel acceleration and intersection techniques for beam tracing. We obtain real-time results for primary rays, faster than the best accelerated ray-tracing methods [RSH05] 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 our 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. 11, we

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, our beam tracer can be 10×–40× faster.

The performance of our method derives from two main technical contributions. We have designed a new algorithm for beam–triangle intersection (Sec. 3.2) 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. We also introduce the first effective method of kd-tree traversal (Sec. 3.3) for beam tracing. While frusta have previously been used to determine a conservative traversal estimate for a bundle of rays (and our approach is inspired by these works) we found that beam tracing can benefit even more than rays from efficient kd-traversal. As a high level decision, we specifically designed both our 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.

Our 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. 4), we 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 our analysis in Sec. 5.1 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 our test scenes), enabling substantial performance improvements over ray tracing. Even in highly tesselated scenes, where the performance benefits relative to ray tracing are reduced, we get exact noise-free soft shadows, independent of resolution in a matter of seconds.

2. Previous Work

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

2.1. Accurate Soft Shadows

Sampling: The sampling-based approach was first introduced as distributed ray tracing [CPC84]. 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 Fig. 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 [ARHM00]. 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 we use in our comparisons in Sec. 5.1, 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. [HDG99] turns point samples into area samples using a flood-fill algorithm. Most methods search for silhouette edges through back-projection and/or tracking visibility events [SG94, DF94]. 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. Our 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 [LAA*05] uses a mixture of the visibility event and sampling approaches. Building upon the idea of shadow volumes [Cro77], 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. [LLA06] 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 [LAA*05] and [LLA06] determine the visible depth complexity from a point, so they still must spend most of their time in a sample integration phase. Our method provides higher accuracy results and raises potential performance by completely removing dependence on point sampling density.

Real-Time Methods: Our work is distinct from approximate real-time techniques [HLHS03]. While these methods can work much faster than our approach or the above algorithms, they focus on plausible rather than accurate soft shadows, requiring significant approximations that break down in specific situations. [SS98], 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 [SKS02, NRH03]. 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. Our method is not comparable, since it only requires precomputation of the kd-tree which has been shown to be interactive even for large scenes [HMS06], easily allows dynamic local lighting, and can be used with general shaders. The recent work

of [RWS*06] 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 our system can handle exact shadows using accurate scene geometry.

2.2. 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. [WSBW01] 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. [RSH05] 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, [WIK*06] 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 our work, we split beams *precisely* at geometry boundaries, exploiting all available coherence with dramatic benefits for both primary and secondary effects.

2.3. Beam Tracing

Beam tracing was introduced in [HH84]. While it has found limited applicability in rendering, it has proven very useful in architectural acoustics [FCE*98, FMC99] where antialiasing is a primary concern. Similarly, our beam-tracing approach provides antialiasing essentially for free, and may therefore also be relevant in this domain.

Two related techniques are cone tracing [Ama84] and ray differentials [Ige99]. 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. [GH98] 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 [TA98] 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 Sec. 2.2 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, our paper removes this assumption, and enables one to explore the full power provided by precise area sampling over point sampling.

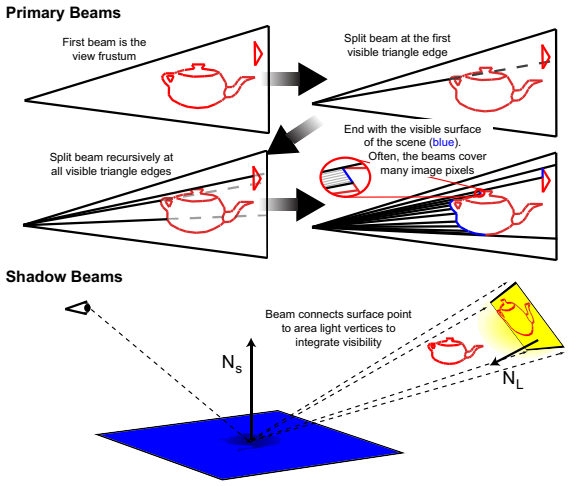

Figure 1: *(Top) Beam tracing for primary visibility. (Bottom) Beam tracing for soft shadows.*

3. Our Beam Tracing Algorithm

For simplicity of exposition, we describe the general beam tracing algorithm in the context of primary visibility. The same ideas extend to secondary shadow beams (see Fig. 1 top and bottom).

As shown in Fig. 1(top), we 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, our scenes are built strictly with triangle primitives, and we 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 [HH84]).

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

3.1. Beam Representation

We represent our 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 [WSBW01]). Pseudocode for our beam representation is in the appendix. It is important to note that we represent the ray directions as points on some plane as this helps with our efficient triangle intersection algorithm described below. For primary beams, we use the image plane. For secondary point light beams, we use the plane of the hit triangle, and for area light beams, we can use a plane on or near the light source. This ensures that accuracy is measured in the relevant space.

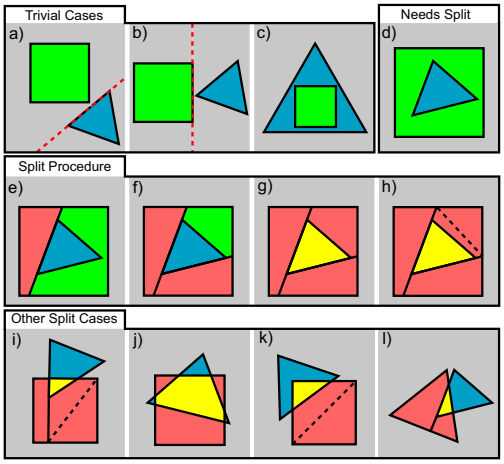

Figure 2: *Illustration of beam–triangle intersection. The initial beam is green, the initial triangle is blue, miss beams are red, and hit beams are yellow. (a)-(c) are trivial cases (step 2). In (d), the beam must split (step 3). (e)-(g) split along each edge of the triangle. In (h), the beam needs 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.*

3.2. 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 [HH84] or off-the-shelf polygon clipping algorithms. Our algorithm is unique and specifically optimized for beam tracing.

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

1. *Mirror, as closely as possible, ray–triangle intersection, diverging only where necessary.* This is aimed at keeping our algorithm as simple as possible, while benefitting from most published ray-triangle intersection optimizations.
2. *Parallelize through the use of SIMD SSE instructions.* While this is not usually viewed as a high-level decision, in our 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, we make the problem easier by solving it in 2D. However, instead of projecting the beam to the triangle's plane (the common ray approach), we project the triangle onto the image plane. This is mostly to avoid numerical 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 we project, we 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 provides an illustration of the 2D beam–triangle intersection problem. There are 3 trivial cases as shown in Figs. 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—we 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, we handle the 4 beam edges in parallel for case (b).

Step 3–Beam Splitting: The rest of the cases fall into Fig. 2d, where some of the beam is inside the triangle and some is outside. The beam now needs to split along triangle edges. We do this using a method like [SH74], who clip one edge at a time. The splitting process is shown in Figs. 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), we are 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. We use a parallel SSE based algorithm which handles all beam vertices at once for each triangle edge (See the appendix 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 we can handle with SSE instructions. In these cases, we perform an extra split (shown as a dashed line in Fig. 2h), generating one 4-corner beam, and one 3-corner beam. The 3-corner beam has a triangular cross-section. We still use 4 corner rays with one being degenerate. Note that Fig. 2d shows a relatively simple situation where the triangle lies fully inside the beam. Figures 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 we use to combat precision errors. First, we 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, we use "fuzzy" logic tests throughout our beam tracer (See the appendix for

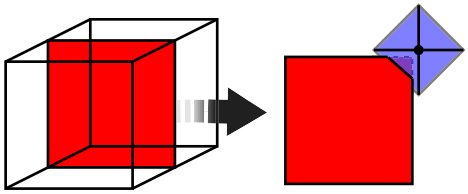

Figure 3: *(Left) The large red plane is the current kd split plane. (Right) the split plane is isolated, the beam is in blue, and our viewpoint is down the axis of the beam pyramid. In this case, the beam's corner rays only want to traverse the far cell, but, clearly, we also need to visit the near cell.*

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) we 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, we next introduce our beam–kd-tree traversal algorithm.

3.3. KD-Tree Traversal

Ray–kd-tree Traversal: The standard ray–kd-tree traversal algorithm as introduced by [Whi80] 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 Fig. 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 ([RSH05] also describes this situation).

Figures 4a-c show our 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) 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 Fig. 3 for 3D. One possible solution would be to split the beam where the kd planes split the far plane as in Fig. 4b. This approach is precise in that it traverses all of and only the cells which intersect the beam. We 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.

Our Solution: Instead, we use the much simpler approach in Fig. 4c inspired by frustum proxy methods like MLRT [RSH05] and LCTS [HB00]. We maintain 3 near and

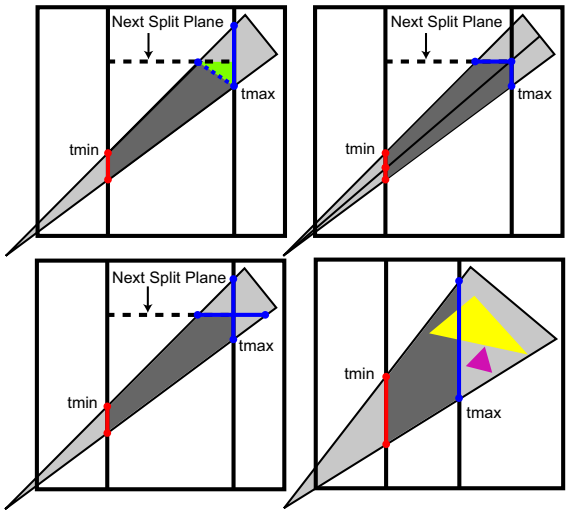

Figure 4: *(a) The minimum and maximum distances for all of a beam's rays cannot be represented by single near and far planes. If we just used the corner rays' far distances, we wouldn't be including the green portion of the beam. (b)We could split the beam with a new far plane. (c) Or we can use multiple (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.*

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 the appendix).

Handling Hit Beams: Our 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 Fig. 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 Fig. 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 we 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 [RSH05]. 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 re-

dundant kd steps. Moreover, the frusta must often visit even more nodes to find a suitably *deep* one.

It is possible that one of our 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 we split *exactly* at geometry boundaries. This substantially reduces redundant kd steps, increases memory coherence, and thus minimizes our performance dependence on absolute scene size.

3.4. 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, we do one processing pass on the triangles before performing any intersection tests. It is during this pass that we 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, we maintain a list of projected triangle vertices. We mark triangles that are either backfacing or fully behind the beam's near plane so that we needn't bother accessing them with later beams in the same frame. We 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. We perform this sorting during the kd-tree build.

Mailboxing For Beams: [WIK*06] 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 our beams since a triangle may have been previously tested against the current beam or any of its parents. To deal with this, we maintain a hierarchical mailboxing structure (we 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. We test against the current beam and its 3 parents. Mailboxing gives us a speed improvement between 10% and 50%.

4. Beam Tracing Analysis–Primary Rays

We 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 our application to soft shadows are referred to Sec. 5.

For our beam tracer, we trace primary visibility, then send our hit beams to the GPU as a list of quads for final rasterization. Since we send full area elements, we 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

Scene / # Triangles / Thumbnail / Average Measurements		Erw6 (816)	Soda Hall (2,195K)	Conference (282K)	Armadillo (345K)
		 180 FPS	25 FPS	4 FPS	0.35 FPS
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 5: *Comparison of statistics for Beams and our 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.)*

per-pixel shading, the GPU portion of our 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 [RSH05]. To focus our comparisons on visibility computation rather than shading, in this section we use only (antialiased) diffuse shading and one light positioned at the camera.

We compare Beams to both our own optimized ray tracer (Rays, Sec. 4.1) as well as MLRT [RSH05] (Sec. 4.2), the fastest current method. We use MLRT's kd-tree builder (with settings tuned specifically for MLRT's performance) for beam tracer, ray tracer, and MLRT alike. We don't use antialiasing for either our 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.

4.1. Beams vs. Rays

Comparison Setup: We compare to our 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—we will verify this by comparing timings with MLRT in Sec. 4.3. We also use this ray tracer later for our comparisons with secondary soft shadow rays in Sec. 5.

The table in Fig. 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), our 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 our 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

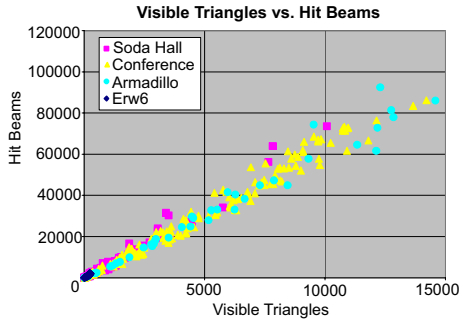

Figure 6: *Number of visible triangles vs. number of hit beams, showing a simple linear relationship. The data points correspond to different views.*

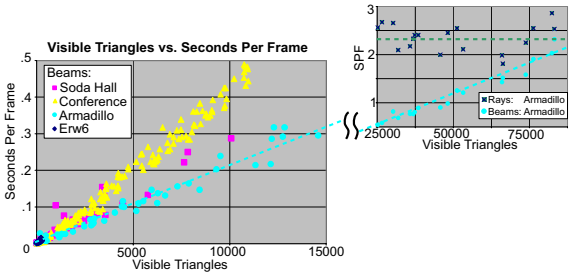

Figure 7: *Number of visible triangles vs. seconds per frame. Running time is proportional to the number of visible triangles for our beam tracer. Different data points correspond to different views of the scenes.*

heavily tesselated Armadillo model is a worst case scenario for our beam tracer with many small triangles which are often smaller than a pixel. Even so, this model shows that our 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 Fig. 6, we 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 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, our 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.

4.2. Beams vs. MLRT

Comparison Setup: We now compare timings against MLRT [RSH05], which is currently the fastest known ray tracer. MLRT uses a frustum proxy mechanism to find deep kd-tree entry points from which it sends 4x4 packets of rays.

To get closer to a direct algorithmic comparison, we set MLRT to use only one thread and turn off quad generation (MLRT aggressively groups axis-aligned triangles into large

# 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 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, we consider the last line for our tests.*

quads allowing the use of a greatly simplified intersection test). We 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 we deal only with beams. We summarize how these optimizations affect MLRT's performance in Fig. 8.

Timing comparisons are shown at the bottom of Fig. 5. Note that these are averages over many views as usual. (They are similar but not exactly the same as in Fig. 8, since the latter were taken from rendering a single view, to enable easier comparison to published MLRT results.) We weren't able to measure the number of kd steps and intersection tests for MLRT, but it is possible to estimate these numbers by taking our ray tracer's measurements (Fig. 5), and adjusting them downward to account for the known improvement MLRT provides. Specifically, to estimate MLRT's kd steps, divide our ray tracer's measurements in Fig. 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 [RSH05].

Performance Comparison: From Fig. 5, we are over an order of magnitude faster than MLRT for Erw6. Our most impressive result is on soda hall, where we are 4× faster. The absolute size of the model slows MLRT down in the kd-tree's inner nodes, while our beams keep to the leaves. While speed is comparable to MLRT for the highly tesselated conference model, we still perform an order of magnitude fewer kd-steps and intersection tests. We perform fewer such tests even on the armadillo where we are 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 our beam tracer, we simply connect our primary hit beams, which often contain 10s to 1000s of pixels, to the point source with another beam. When we trace this beam, the resulting hit list represents obstructed portions of the beam, while the miss list is fully unobstructed.

The images in Fig. 9 were rendered with our beam tracer at 1024x1024 inside the soda hall model at 21 FPS. MLRT can only render this view at 5 FPS. The left image shows our beam tracer's high quality shadows, while the right image shows the beams which created it—notice that our

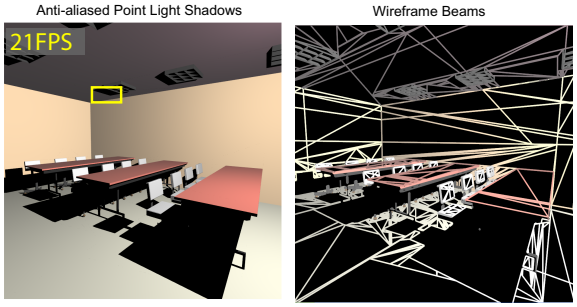

Figure 9: *Real-time antialiased point light shadows. (Left) This image was rendered at 21 frames per second at 1024x1024 resolution with 6× antialiasing inside the 2 million triangle soda hall model. (Right) The wireframe beams which generated the top-left image: all geometry and shadows are cut out exactly providing for high-quality antialiasing (Fig. 10).*

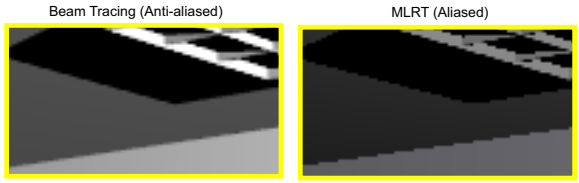

Figure 10: *(Left) close-up of highlighted region from Fig. 9. We render with 6× antialiasing with almost no performance penalty. (Right) same region with MLRT clearly displaying jagged edges and spurious black pixels.*

beam tracer produces a perfect shadow cut-out for high quality antialiasing. We display a close-up of the yellow highlighted region for both beam tracing and MLRT in Fig. 10. The MLRT image (right) clearly shows jagged object and shadow edges along with spurious black pixels. Our image is rendered with 6× antialiasing with almost no performance penalty.

4.3. Rays vs. MLRT

Before moving on to soft shadows, we take a moment to compare MLRT to our own ray tracer, since we use the latter for comparisons in Sec. 5. From the bottom of Fig. 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.

We therefore conclude that our ray-tracer is quite well optimized for a one-at-a-time tracer. 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 Sec. 5.2). Therefore, we use our heavily optimized ray tracer as the comparison method in the next section.

5. Soft Shadows

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

$$B(\mathbf{x},\omega_o) = \int_A V(\mathbf{x},\mathbf{p}) L(\omega_i)\rho(\omega_i,\omega_o)\cos\theta_i\cos\theta_l\, d\mathbf{p}, \quad (1)$$

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, we simply create a beam whose apex is the image point and base is a triangle or quad of the light source (See Fig. 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 we choose one based on which list is smaller.

For simplicity of comparisons in Sec. 5.1, we integrate the lighting using the common approximation also used in [SS98] and [ARHM00]. We 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,

$$B(\mathbf{x},\omega_o) = L\rho(\omega_i,\omega_o)\cos\theta_i\cos\theta_l \int_A V(\mathbf{x},\mathbf{p})d\mathbf{p}, \quad (2)$$

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 [HDG99] for Lambertian surfaces or [Arv95] for specular. However, we found that equation 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.

5.1. Results

Comparison Setup: To render an area lit image, we first need to trace primary visibility for which we use our 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).

Our 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. We do so here to provide a context to measure our performance. As such, determining the correct number of samples and sampling strategy for "comparable quality" results is somewhat arbitrary. We use 256 samples on a jittered grid as this is often considered the minimum requirement to produce high quality soft shadows.

We compare to our own optimized ray tracer As we showed in Sec. 4.3, our ray tracer is quite well optimized.

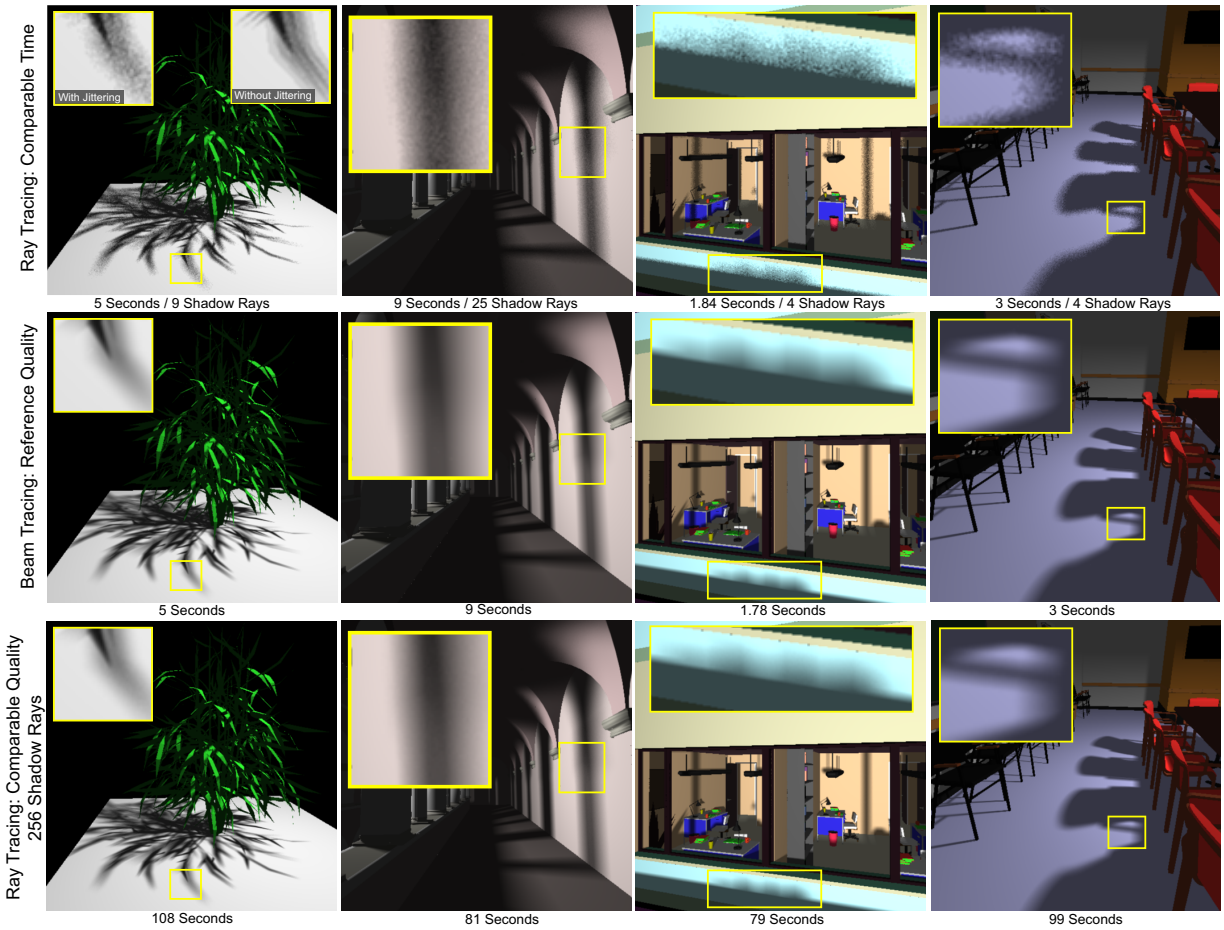

Figure 11: *Beam Tracing vs. Ray Tracing. Beam tracing (center row) provides an exact lighting solution in seconds. Ray tracing requires 256 samples or shadow rays (bottom) to reduce noise within tolerance. Attempting to ray trace with few enough samples to match the speed of beam tracing (top) leads to severe noise. All images are rendered at a resolution of* 512×512 *pixels.*

We do not use MLRT because it does not provide a suitable area lighting implementation. Besides, as described in Sec. 5.2, we do not believe it would provide significant benefits. Since we seek to study the effectiveness of secondary visibility determination, we use Lambertian materials and a single square light in all comparisons. We use our 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 [Hav00], and the nodes use a cache optimized layout as in [WSBW01].

Scenes: The plant scene has 5245 triangle faces. Almost every edge is a silhouette edge making it difficult for edge based methods such as [LAA*05, LLA06]. 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 11 compares image quality and wall clock time using our beam tracer vs. ray tracing. The center row, generated using our beam tracer, serves as the reference result since we 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, we 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, our beam tracer achieves a $10\times$–$40\times$ improvement over ray tracing for "comparable" image quality.

Quantitative Comparison: Figure 12 shows several statistics for beam tracing and ray tracing (with 256 shadow rays). We divide all statistics by the image resolution (512x512) to give per pixel measurements. We also include the number of visible triangles seen from each pixel by our beam tracer for the shadow beams, since the number of visible triangles was identified as a primary scene component relating to our performance in Sec. 4.

We are 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, our beam tracer can operate on hundreds of thousands of image pixels per second with the highest accuracy.

Also note that secondary beams are able to process many more triangles per second than primary beams (compare visible triangles per second in Figs. 5 and 12). Recall that for

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 12: *Performance of beam tracing for soft shadows. We show several statistics for generating the images in Fig. 11. We compare the performance of our beam tracer with our ray tracer (the latter with 256 shadow rays). Our beam tracer uses significantly less kd steps and intersection tests leading to at least an order of magnitude improvement.*

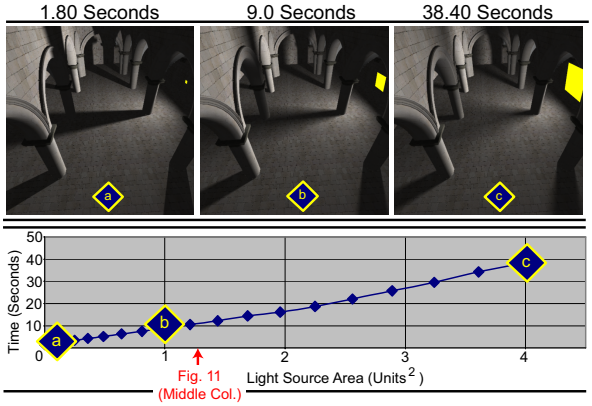

Figure 13: *Beam tracing scalability with light source size. Our beam results are nearly linear in the light source size.*

primary visibility, the number of hit beams was about 6× the number of visible triangles (Fig. 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, we 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, we benefit even more by being able to quickly eliminate more of the light source.

Light Source Area vs. Time: Figure 13 shows light source area versus wall clock performance using our 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. Our 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 our ray tracer renders these images in around 94-98 seconds with 256 shadow rays, so even for a very large light source, our 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, our beam tracer always produces exact results, never needing to point sample the light source.

5.2. 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 we showed that MLRT can be 10×–14× faster than our ray tracer for primary visibility, we don't believe it would be nearly as efficient for soft shadows. When calculating primary visibility, we are 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 we demonstrate.

Limitations: Our 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 [LLA06] may provide faster results, but cannot guarantee the same quality since we 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 our 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, we 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.

6. Conclusions and Future Work

We 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, we compute exact and noise-free soft shadows in a matter of seconds.

We have only begun to optimize our beam tracing implementation. The results in Fig. 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, we would also like to exploit image space coherence. We 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. Gen-

erating specular reflections and caustics is one promising direction. In fact, the work of [LWY*07], developed concurrently with ours, 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. Our work proves that this perception is largely undeserved, and we have only provided a peek into the true potential for high speed beam tracing. We hope this work encourages more attention in this promising direction.

Acknowledgements

This research was funded in part by NSF grants #0305322, #0446916, and #0546236, a research grant from Intel Corporation, a Sloan Research Fellowship, and an ONR Young Investigator Award N00014-17-1-0900. We would also like to thank Alexander Reshetov for MLRT code and scenes, and Aner Ben-Artzi and Kevin Egan for many helpful discussions. The Sponza Atrium model is provided courtesy of Marko Dabrovic.

References

[Ama84] AMANATIDES J.: Ray tracing with cones. In *SIGGRAPH 84* (1984), pp. 129–135.

[ARHM00] AGRAWALA M., RAMAMOORTHI R., HEIRICH A., MOLL L.: Efficient image-based methods for rendering soft shadows. In *SIGGRAPH 00* (2000), pp. 375–384.

[Arv95] ARVO J.: Applications of irradiance tensors to the simulation of non-Lambertian phenomena. In *SIGGRAPH 95* (1995), pp. 335–342.

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., SHIRLEY P., WALD I.: Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface* (May 2007).

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. In *SIGGRAPH 84* (1984), pp. 137–145.

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. In *SIGGRAPH 77* (1977), pp. 242–248.

[DF94] DRETTAKIS G., FIUME E.: A fast shadow algorithm for area light sources using backprojection. In *SIGGRAPH 94* (1994), pp. 223–230.

[FCE*98] FUNKHOUSER T., CARLBOM I., ELKO G., PINGALI G., SONDHI M., WEST J.: A beam tracing approach to acoustic modeling for interactive virtual environments. In *SIGGRAPH 98* (1998), pp. 21–32.

[FMC99] FUNKHOUSER T. A., MIN P., CARLBOM I.: Real-time acoustic modeling for distributed virtual environments. In *SIGGRAPH 99* (1999), pp. 365–374.

[GH98] GHAZANFARPOUR D., HASENFRATZ J.-M.: A beam tracing method with precise antialiasing for polyhedral scenes. *Computers and Graphics 22*, 1 (1998), 103–115.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HB00] HAVRAN V., BITTNER J.: LCTS: Ray shooting using longest common traversal sequences. *Computer Graphics Forum 19*, 3 (2000), 59.

[HDG99] HART D., DUTRE; P., GREENBERG D. P.: Direct illumination with lazy visibility evaluation. In *SIGGRAPH 99* (1999), pp. 147–154.

[HH84] HECKBERT P. S., HANRAHAN P.: Beam tracing polygonal objects. In *SIGGRAPH 84* (1984), pp. 119–127.

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F.: A survey of real-time soft shadow algorithms. *Computer Graphics Forum 22*, 4 (Dec. 2003), 753–774. State-of-the-Art Reviews.

[HMS06] HUNT W., MARK W. R., STOLL G.: Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[Ige99] IGEHY H.: Tracing ray differentials. In *SIGGRAPH 99* (1999), pp. 179–186.

[LAA*05] LAINE S., AILA T., ASSARSSON U., LEHTINEN J., AKENINE-MÖLLER T.: Soft shadow volumes for ray tracing. *ACM TOG SIGGRAPH 05 24*, 3 (2005), 1156–1165.

[LLA06] LEHTINEN J., LAINE S., AILA T.: An improved physically-based soft shadow volume algorithm. *Computer Graphics Forum 25*, 3 (2006), 303–312.

[LWY*07] LIU B., WEI L., YANG X., XU Y., GUO B.: *Nonlinear Beam Tracing on a GPU*. Tech. Rep. MSR-TR-2007-34, Microsoft Research, 2007.

[NRH03] NG R., RAMAMOORTHI R., HANRAHAN P.: All-frequency shadows using non-linear wavelet lighting approximation. *ACM TOG SIGGRAPH 03 22*, 3 (2003), 376–381.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM TOG SIGGRAPH 05 24*, 3 (2005), 1176–1185.

[RWS*06] REN Z., WANG R., SNYDER J., ZHOU K., LIU X., SUN B., SLOAN P.-P., BAO H., PENG Q., GUO B.: Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. *ACM TOG SIGGRAPH 06* (2006), 977–986.

[SG94] STEWART A. J., GHALI S.: Fast computation of shadow boundaries using spatial coherence and backprojections. In *SIGGRAPH 94* (1994), pp. 231–238.

[SH74] SUTHERLAND I. E., HODGMAN G. W.: Reentrant polygon clipping. *Commun. ACM 17*, 1 (1974), 32–42.

[SKS02] SLOAN P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM TOG SIGGRAPH 02 21*, 3 (2002), 527–536.

[SS98] SOLER C., SILLION F.: Fast calculation of soft shadow textures using convolution. In *SIGGRAPH 98* (1998), pp. 321–332.

[SWZ96] SHIRLEY P., WANG C., ZIMMERMAN K.: Monte carlo techniques for direct lighting calculations. *ACM TOG 15*, 1 (1996), 1–36.

[TA98] TELLER S., ALEX J.: *Frustum Casting for Progressive, Interactive Rendering*. Tech. Rep. MIT/LCS/TR-740, 1998.

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

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM TOG SIGGRAPH 06* (2006), 485–493.

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164.

Appendix A: Algorithm Details

In the following subsections we give low-level details and pseudocode for beam–triangle intersection and kd-tree traversal.

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
};
```

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. 14b? If we were dealing with rays (Fig. 14a), this is a difficult question to answer, but when dealing with beams (Fig. 14b), 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));
```

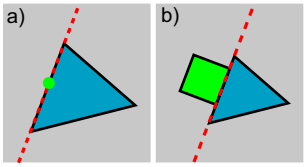

Figure 14: *A difficult intersection test case requiring fuzzy logic. Does the infinitesimally small ray in (a) (green) lie inside or outside the triangle (blue) as determined by its left edge (red)? Clearly the beam in (b) (green) misses the triangle, but floating point accuracy may tell us otherwise.*

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

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 SplitSubO = Split - CurBeam->Origin;
      float4 Dist      = CurBeam->InvDirs[axis] * SplitSubO;
      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);
  }
}
```