Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing

Carsten Benthin Ingo Wald

Intel ® Corporation

Abstract

Ray tracing has long been considered to be superior to rasterization because its ability to trace arbitrary rays allows for simulating virtually any physical light transport effect by just tracing rays. Yet, to look plausible, extraordinary amounts of rays for effects such as soft shadows are typically required. This makes the prospects of real-time performance rather remote. Rasterization, in contrast, has a record of producing such effects in real-time through employing specialized and approximate solutions for individual effects. Though ray tracing may still be the right choice for effects like reflections and refractions, using specialized solutions for certain important effects also makes sense for a ray tracer.

In this paper, we propose a special solution to ray trace soft shadows that is particularly targeted for Intel's Larrabee architecture. We use a specialized frustum tracing that traces multiple frusta of specialized "light-weight" shadow packets in parallel, while generating rays within each frustum on demand. The technique can easily be integrated into any packet ray tracer, and fits well into the wide SIMD and cache-size constraints of the Larrabee architecture. Our technique allows to reach rates of up to several dozen million rays per second per Larrabee core, outperforming traditional packet techniques by up to 6×. This high performance combined with a simple light-weight illumination filtering step allows to achieve real-time soft shadows for game-like scenes.

1 Introduction

Real-time graphics today is almost exclusively based on Z-Buffer technology. Yet, ray tracing is often considered as a possible future alternative that might eventually lead to higher image quality and increased ease of content creation. This is based on the ability to virtually simulate every physical lighting effect through tracing rays and to easily combine all these effects in a single rendering framework. This *generality* is important from the content creation side, and may - eventually - well be the deciding argument in favor of ray tracing.

Today, however, ray tracing is still far too slow to compete with rasterization for real-time applications like games. Interestingly, it is actually the image quality that usually argues in favor of Z-Buffer based techniques for games: improvements in hardware and traversal algorithms nowadays allow for real-time ray tracing performance for primary visibility and hard shadows, but as soon as effects like soft shadows, motion blur, anti-aliasing, ambient occlusion, etc. are added, a ray tracing based renderer can no longer maintain real-time frame rates due to the vastly increased number of rays required to compute these effects. While Moore's law argues that it is only a matter of time until hardware will be fast enough to trace even these numbers of rays in real time, this is certainly not yet the case.

Using rasterization, in contrast, most of these effects are (under proper circumstances) affordable at real-time rates, and are commonly used in today's games running on current hardware. This is because programmers usually use specialized solutions for these effects, such as shadow maps [Soler and Sillion 1998; Fernando 2005; Johnson et al. 2009] and shadow volume based techniques [Assarsson et al. 2003] for soft shadows, screen-space filtering techniques for ambient occlusion and motion blur effects, multi-sample anti-aliasing (MSAA) etc. These special solutions often have problems of their own, but usually are orders of magnitude more efficient than computing the same effect through tracing rays.

The drawback of using such techniques is that it is often problematic to combine them with each other, and they often rely on properly modeled scenes and manual software-tuning. Nevertheless, for today's games this is usually not a deal-breaker, and if ray tracing is ever to compete with rasterization in the real-time domain, either massively more powerful hardware is needed, or similarly specialized solutions for the most important effects have to be investigated for ray tracing too. Ideally, such special solutions should give similar benefits as they did for rasterization, but not negatively impact the ray tracer's plug-and-play characteristics.

In this paper, we investigate a special solution for ray traced soft shadows for game-like scenes. Instead of tracing dozens of "standard" rays per pixel to compute a soft shadow, we trace shadow frusta from the hit points to the light source and generate rays for intersection tests on demand. After tracing all shadow frusta, the irradiance at each pixel is stored. In the final step a small filter kernel is applied to the entire image to filter neighboring irradiance.

For traversal and intersection, we use specially designed data layouts, culling techniques, and optimized algorithms that provide dramatically higher performance than packet tracing approaches. Though the technique is particularly designed for soft shadows, it can also be applied efficiently for primary visibility (e.g., for multi-sample anti-aliasing) or hard shadow rays. It cannot accelerate other rays like reflection or refraction rays at all, however, it can be easily integrated into an existing packet-based ray tracer without major modifications (i.e., it also works for soft shadows seen by reflected and refracted rays). Furthermore, it does not require additional information like silhouette edges by itself, but could potentially make use of it. In addition, our technique is specially designed to fit well into the architectural constraints—in particular, wide SIMD and limited amounts of per-thread storage/cache—that are the hallmark of modern high-thoughput architectures. Though also applicable to similar architectures, our particular implementation is optimized for the Larrabee architecture [Seiler et al. 2008].

Our technique is able to generate ray traced soft shadows up to 6× faster than computing the same image with packet tracing. Using a cycle-accurate Larrabee simulator—configured, like in [Seiler et al. 2008], to simulate a hypothetical clock rate of 1 GHz—we demonstrate that 16 cores would suffice to reach more than half a billion primary and soft shadow rays per second (and thus, real-time performance).

2 Background

2.1 The Larrabee Architecture

Larrabee [Seiler et al. 2008] is a scalable multi-core design that comprises a group of small in-order processor cores (and certain fixed-function units, see Figure 1) all of which are connected by a high-throughput ring bus. Each core is derived from the Intel Pentium processor, providing full support for the Pentium x86 instruction set. In addition, each core features a 512-bit wide SIMD unit that performs sixteen 32-bit or eight 64-bit integer and floating point operations.

All vector operations can be controlled by 16-bit mask registers, which allows for handling SIMD divergence (rays not taking a certain code path can easily be disabled by setting their mask bits to 0). In addition to the typical arithmetic and logical vector operations the *LRBni* instruction set [Intel LRBni 2009] also supports free conversions for various data formats, as well as three-operand instructions, element swizzling, scatter/gather operations, and even special operations such as bitcounts, bitscans, etc.

Larrabee cores are in-order, but run four different hardware threads in parallel. Each core has its own sets of (fully coherent) 32 KB L1-cache and 256 KB of L2 cache, with communication between processors done by the ring bus. The cache hierarchy is fully coherent among all cores.

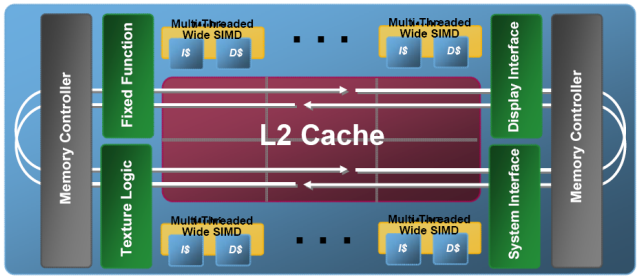

Figure 1: The Larrabee architecture consists of multiple small in-order x86 processor cores. Each core has its own L1 cache and L2 cache, 4 hardware threads, and a 16-wide vector unit.

2.2 Packet and Frustum Traversal Techniques

Before SIMD became ubiquitous, achieving high ray tracing performance relied on tracing individual rays through an optimized spatial index structure. In the presence of SIMD, operating on individual rays becomes problematic. To solve that problem, *packets* of rays are traced together [Wald et al. 2001], typically using packet sizes equal to the SIMD width. This approach can also be used on Larrabee, and will provide the baseline for our comparisons.

In general, the benefit of traditional packet tracing is limited by the SIMD width. Greater speedups can be achieved by using even larger packets, and bounding them by frusta, envelopes, or interval arithmetic. This bounding information can then be used to effectively cull entire packets during traversal and/or intersection [Dmitriev et al. 2004; Reshetov et al. 2005; Boulos et al. 2006; Wald et al. 2007].

Rather than using the bounding information only for culling, one can also base the entire traversal exclusively on this bounding information [van der Zwaan et al. 1995; Wald et al. 2006; Overbeck et al. 2007]. Since the bounding information has to be conservative, this may lead to some unnecessary traversal steps, but since the bounding tests are typically cheap compared to testing multiple rays, a significant speedup can usually be achieved. Unfortunately, such traversal techniques often do not fit well to wide-SIMD architectures, as we will discuss in more detail in Section 3.

2.3 Soft Shadow Algorithms

The first approaches for computing soft shadows used ray tracing for Monte-Carlo-based sampling [Cook et al. 1984]. In order to reduce the noise caused by variance in the estimate, a large number of samples (> 256) was required. Exact soft shadows are obtained by determining the precise occluder geometry by either turning a flood-fill algorithm on point samples [Hart et al. 1999] or searching for silhouette edges [Stewart and Ghali 1994; Drettakis and Fiume 1994]. These approaches produce high quality soft shadows but are typically slower than sampling based techniques.

Soft shadow volumes [Laine et al. 2005; Lehtinen et al. 2006] utilize penumbra wedges to determine the screen space of potential silhouette edges. Both approaches require memory-intensive data structures and complex query algorithms, which do not fit well into the Larrabee architecture. The same problems apply to beam techniques [Overbeck et al. 2007] as polygon-clipping and frustum splitting require complex state handling, making it hard to efficiently map them to a wide-SIMD architecture.

Approximate techniques focus on generating plausible instead of accurate soft shadows [Hasenfratz et al. 2003; Amanatides 1984; Soler and Sillion 1998; Parker et al. 1998]. These approaches tend to break down in certain situations (e.g. self-shadowing), require manual parameter tuning, or demand significant preprocessing (as, for example, in the case of precomputed radiance transfer [Sloan et al. 2002]).

Today's fastest soft shadow algorithms either use light-view based spatial index structures [Johnson et al. 2009] or pre-filtered shadow maps [Annen et al. 2008]. They are very fast and produce plausible soft shadows which might not be physically accurate. However, they are purely rasterization-based and are not easy to integrate into a pure ray-tracing based rendering system.

On the ray tracing side, Instant Radiosity [Keller 1997] in combination with interleaved sampling [Keller and Heidrich 2001], discontinuity filtering, and packet tracing, has been shown to compute indirect illumination at interactive rates [Wald et al. 2002]. Our soft shadow approach uses these techniques, in particular interleaved sampling and a light-weight discontinuity buffer, to significantly reduce the number of required shadow rays per pixel (compared to standard sampling).

3 Multi-Frusta Tracing (MFT)

Disregarding memory access, the performance of a packet ray tracer strongly depends on the SIMD utilization, which is particularly true for Larrabee's wide-SIMD. As mentioned in Section 2.2, ray packet tracing in general fits well to a SIMD architecture, but higher speedups can only be achieved by using bounding information to handle large sets of rays. Obviously, the bounds need to enclose the contained rays tightly, which means that the rays themselves need to be coherent. In the following, we will refer to the bounding information simply as *frustum*.

The major drawbacks of frustum techniques are that the higher the SIMD width, the lower the potential speedup over packet tracing and the more difficult it is to make efficient use of SIMD. For example, the fast BVH traversal algorithm by [Wald et al. 2007] performs essentially two scalar tests: the *first hit* test and the *frustum culling* test. These two (logically scalar) tests can still be mapped reasonably well to 4-wide SIMD, but not to 16-wide SIMD any more. Additionally, for the proposed packet size of 64 rays [Wald et al. 2007], the potential maximum traversal speedup, compared to 16-wide packet tracing, will be $4\times$ (instead of $16\times$ for 4-wide SIMD). Getting the same theoretical benefits would require to use even larger packets, which however leads to other problems like increased cache footprint, higher coherency demands, etc.

3.1 Our approach

Instead of performing a single frustum culling test at a time, our approach does the culling for 16 frusta in parallel. As the culling is independent across the frusta, it maps well to 16-wide parallel SIMD execution. Our multi-frusta tracing (*MFT*) uses interval arithmetic-based culling [Wald et al. 2007] of axis-aligned bounding boxes (AABB). Extensions to other BVH variants (e.g., oriented bounding boxes or spheres) are straightforward. Rather than interval arithmetic one could also use bounding planes, bounding cones, etc, but our interval-based approach is both simple and efficient. Note that we use "open ended" frusta without a far plane, as this has shown to be eaiser and faster than trying to save some some traversal steps by maintaining such a far plane.

Compared to the algorithm by Wald [2007], we drop the *first hit* test and the *packet-intersection fallback* test. The latter computes ray-AABB intersections for all active rays and is executed every time the *first hit* and the *interval-based culling* tests fail. The *first hit* and *packet-intersection fallback test* are the only tests that require individual rays. Our BVH traversal relies exclusively on the culling efficiency of the 16 frusta. This in particular requires extensions to the culling itself (see Section 4.1) and to the primitive intersection test at the leaf level (see Section 4.2). Compared to tracing only a single 'big' frustum at a time [Reshetov et al. 2005], multiple small frusta are more efficient as they can be terminated individually during traversal, which is similar to packet traversal where individual rays can be terminated.

Our *MFT* technique does not require major changes to the underlying ray tracer; instead, it is built on top of an already existing ray tracing core. The core's internal data structures for storing scene data are simply reused. Though also applicable to other data structures, for the rest of this paper we assume that the underlying ray tracer uses a bounding volume hierarchy (BVH). We also assume that this BVH has already been built/updated for this frame, but extensions to lazy construction schemes are straightforward.

3.2 Usage scenarios

MFT is not tied to one particular shading/lighting algorithm, but rather a general tool that can be used in various ways. Though one could in theory build an entire ray tracing system around *MFT*, we do not recommend this: as any other frustum-technique MFT depends on the rays in each frusta to be coherent, and maintaining sufficient coherence for arbitrary secondary rays is not trivial.

Instead, we suggest to trace arbitrary rays with traditional packet tracing, and use our approach only for specific kinds of rays that are sufficiently dense and coherent. For now, we focus only on primary rays (16 frusta of 16 rays) and soft shadows (16 shadow rays each for a packet tracer's 16 surface samples), but will briefly sketch other applications in Section 6.1. Concentrating on specific kinds of rays also allows further optimizations like early frustum termination and on-demand generation of the rays, which we will detail below.

4 Implementation

4.1 Traversal

Our BVH traversal is based on culling AABBs using interval arithmetic. As a common origin is assumed for each frustum (but not for all frusta), only intervals over the ray directions are required, more precisely over the reciprocals of ray directions per frustum. Each AABB is stored using a 32-byte layout, while the AABBs for the left and right child of a BVH node are stored consecutively in a 64-byte block (a single cache line).

The culling step itself performs a 3D slabs-test [Kay and Kajiya 1986] using intervals [Boulos et al. 2006], which is done for all 16 frusta in parallel. It first subtracts the origin from min/max values of the AABB and then computes the intersection of interval bounds per dimension (using the reciprocal direction intervals). In case all frusta share a single origin (primary or hard shadow rays), only a single subtraction is needed for the two AABBs (left and right child). The code complexity of 16-wide interval-based slabs test is similar to a standard 16-wide ray slabs test.

During traversal of the BVH, we keep track of all active frusta by simply tracking an *active mask*. If a frustum has been terminated, it is marked as *inactive* (by setting the corresponding bit in the mask to zero), after which this frustum will not be considered in further traversal/intersection steps.

As multi-frusta traversal does not consider individual rays, only the triangle intersection test itself requires them. This avoids pre-traversal setup such as computing and storing ray directions and their reciprocals. The latter are typically required for an optimized slabs test. The drawback of performing only interval-based culling without a per-packet fallback is the dependence of culling efficiency to the extent of the interval bounds.

Even for tight ray direction intervals, the culling efficiency breaks down if one of them contains zero in any of the three dimensions. In this case, the reciprocal interval is the full interval $(-\infty, +\infty)$, meaning a complete loss of culling efficiency in the respective dimension. In order to fix it, a simple additional culling step is introduced: if the ray directions within a frustum differ for a given dimension, an intersection test between the AABB of the ray directions and the node's AABB is performed. This assumes that the interval in the zero-containing dimension is tight, which is usually the case. Moreover, the case of different directions per dimension is far less likely than the common direction case, e.g. less than 4% for primary rays.

For all test scenes used, the culling-based traversal increases the number of active frusta reaching leaves only slightly, compared to performing exact ray-AABB intersection tests at the leaf level. The increased number of triangles passed to the intersection test can be efficiently accommodated by fast triangle culling techniques, see Section 4.2. In general, all our application scenarios, e.g. primary visibility and soft shadows, assume that the frusta are reasonably tight, meaning the rays in frusta are somewhat coherent. Otherwise, frustum shrinking during traversal can be applied, see Section 5.3.

4.2 Intersection

As described so far, MFT can only reduce the number of AABB tests. However, the same concept can also be used during (multi-)packet-triangle intersection, by performing some additional "per packet" culling tests (like backface culling and corner ray culling) for all 16 packets in parallel. In total, once reaching a leaf we iterate over all contained triangles and perform up to three tests per triangle, see Figure 2.

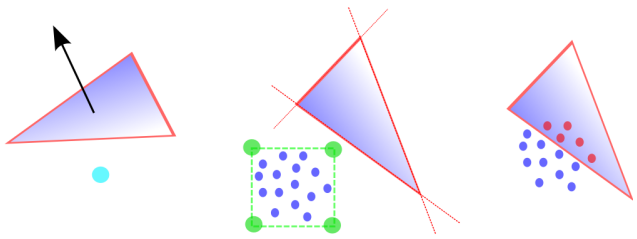

Figure 2: Left: The first test checks the orientation of the triangle with respect to the frustum's origin. If the triangle's normal has the wrong orientation the triangle is culled. Center: Testing to separate the four corner rays (which bound the frustum) from each of the three triangle edges. If the four corner rays are *outside* with respect to one of the edges, the triangle is culled too. Right: Final ray-packet triangle intersection test, by first checking the aperture and then the distance.

First, we check the orientation of the triangle with respect to the frustum origin (back-face culling) in parallel for all 16 frusta. After and'ing with the active frustum mask, we know which frusta are still active after backface culling, and can exit if no frustum remains active. This test is particularly useful to avoid intersections with the triangles on which the shadow rays originate.

Second, we perform a more accurate triangle vs. frusta separation test. We use the four 'corner' rays of each frustum to determines whether any frusta are entirely 'outside' any of the triangle's edges [Boulos et al. 2006; Wald et al. 2007]. This test is again performed for all 16 frusta in parallel, and the result mask is logical ANDed with the back-face culling mask and the active frusta mask found during traversal. The final mask, which we call *triangle active mask*, indicates for which frusta a full 16-wide ray-triangle intersection test has to be performed. The corner ray test is efficient in compensating the drawbacks of the culling-based traversal, as it quickly culls a large fraction of non-intersecting triangles, see Section 5. It mostly offsets any potential benefit from full ray packet-AABB intersection tests at the leaf level.

Third, we scan over all active frusta (using bitscan) and performs a 16-wide ray packet-triangle intersection (only) for the active frusta. Note that all operations so far (AABB tests, backface culling, corner ray tests) have considered only the bounding frusta, so this packet/triangle test now is the only point where we actually have to consider the individual rays. Rather than reading pregenerated rays from memory, for primary and shadow rays it is actually faster to generate these rays on demand (as detailed Section 4.4). In particular, for fully lit regions this means that in most cases no shadow rays are ever generated, as in those cases the only tests performed are AABB tests, back-face culling, and possibly corner-ray culling, all of which use only the frustum information.

For packet/triangle intersection, we use a variant of the Pluecker test that allows to exploit the common origin per frustum and to reuse computed data between all three tests [Benthin 2006]. By first subtracting the origin(s) from the three vertices and computing a cross product per edge, the triangle aperture test (for corner and frustum rays) simplifies to three SIMD dot products. Obviously, this setup is performed in SIMD for all 16 frusta in parallel.

Each dot product between a ray and a triangle edge determines their orientation. Since we perform back-face culling, the triangle orientation is known, so we only have to check for the 'correct' sign of each dot product. The corner ray culling step is done the same way, by just checking the orientation of the four corner rays. If all four rays have the 'wrong' orientation, no intersection occurs.

For the first two tests, early exits are implemented: if, for example, a triangle has a wrong orientation with respect to all active frusta, it will be culled. For the actual triangle intersection test, the aperture test is performed before the distance test, as it is likely to have a higher culling rate than the distance test.

It is worth mentioning that no preprocessed triangle data is used. The underlying ray tracing core just stores three vertex indices per triangle. Storing some precomputed per-triangle data leads to even higher performance (by reducing the number of memory accesses), but is not done in our experiments below.

4.3 Ray Termination

Shadow rays can be terminated as soon as any intersection is found. Similarly, a shadow frustum can be terminated as soon as all contained rays report an intersection. For these frusta the respective bits in the active mask are set to zero, and no further triangle intersection tests will be performed. Compared to tracing one big primary ray frustum at a time, tracing multiple smaller frusta thus has four big advantages: it better utilizes SIMD units by operating on 16 different frusta at a time; it allows to perform common origin optimizations (eg, in the triangle test) as long as all rays in any given frustum have the same origin; it allows to selectively terminate individual frusta while other frusta may still be active; and finally, it leads to better culling efficiency because the 16 individual frusta are much tighter than one bug frustum encompassing all rays (see Figure 4).

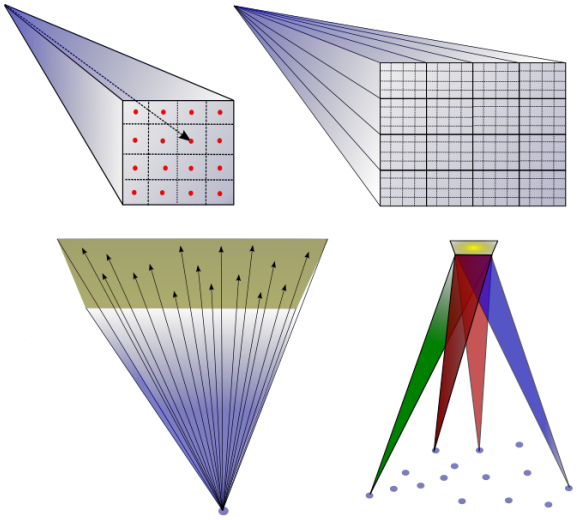

Figure 4: Top left: Single primary ray frustum, representing a 4×4 pixel grid. Top-Right: 16 primary ray frusta, grouped in a 4×4 grid, representing 16×16 pixels. Bottom-Left: Single shadow ray frustum, enclosing 16 shadow rays, starting from a common origin to 16 samples on the light source. Bottom-Right: 16 shadow ray frusta, enclosing 256 shadow rays. Each shadow ray frustum has a different origin. Note that the 16 individual shadow frusta are much tighter than a single frustum spanning all shadow rays.

4.4 Frusta Generation and Data Layout

Primary Ray Frusta Each primary ray frustum corresponds to a 4×4 pixel grid. All 16 frusta represent 16×16 pixels or 256 rays, see Figure 4. Generating corner rays for the 16 primary frusta by using the image plane is trivial, as well as computing the respective ray intervals from the corner rays. As storing data for 256 rays increases the cache-footprint by more than 3KB (+3KB for reciprocals), it is more beneficial to generate rays quickly on demand. Therefore, we decompose each ray direction into two 3D vectors: a *start* and *delta* vector. The *start* vector corresponds to the upper-left corner ray of each frustum while the *delta* vector represents 16 offsets to the centers of a 4×4 pixel grid. Generating the 16 rays for frustum n can now be done by loading the *delta* vector and adding the replicated n entry of the *start* vector. This is done for the 16 rays in parallel, requiring only 3 SIMD *loads* and 3 SIMD *adds*, due to the free broadcast feature of the LRBni instruction set. As the *delta* vector for primary rays is constant over all 16 frusta, it is only loaded once per leaf.

The traversal only needs the frusta origin(s) and the reciprocals of the 16 ray directions intervals: one min and max value per dimension yield a total of 6 SIMD vectors. Triangle intersection tests require the four corner rays per frustum and the *delta* vector (the *start* vector is the upper-left corner ray) to create the rays within each frustum. The four corner rays require $4 \times 3 = 12$ SIMD vectors. By using *delta* vectors, the four corner rays can be created on demand too, which would require only 3 SIMD vectors plus the respective delta vector.

As mentioned in Section 3.2, complete data has to be computed and stored only for intersection. Each *hit point* typically requires data such as triangle ID, barycentric coordinates, distance, vertex indices, geometric and shading normal etc. Intersection data for $16 \times 16 = 256$ rays in particular requires that a large fraction (> 12 KB) of the L1 cache is dedicated to it. Handling even more than 16

Figure 3: Test scenes: **T-Rex** (69K triangles), **Fern** (212K triangles), **Saloon** (60K triangles) taken from the game "Call of Juarez" by Techland, and **Fairy Forest** (174K triangles). All scenes are rendered at 1024×1024 resolution with multi-frusta tracing and 16 shadow samples per pixel. Samples are generated by interleaved sampling and filtered using a 4×4 discontinuity filter.

frusta at a time would further increase the pressure on the L1 cache, making it unlikely to achieve any significant speedups.

Soft Shadow Frusta For 16 hit points, 16 frusta are traced to the light source (see Figure 4). We currently assume quadrangular lights; extensions to other light source shapes are straightforward.

As each frusta has a different origin, 16 'different' origins are used during traversal and intersection. However, no intersection data per shadow ray is required, as we only track a binary state ('occluded' or 'not occluded'). For less than 32 shadow rays per frustum, the binary states of all shadow rays (≤ 512 bits) can be stored in the bits of a single SIMD vector. Additionally, 3 SIMD vectors for origin, 6 SIMD vectors for interval reciprocals, 12 SIMD vectors for the four corner rays are required. All shadow rays are created on demand using the 16 pre-computed *delta* vectors and one of the corner rays. Compared to primary frusta, shadow frusta are very 'cache-friendly' as only a very small amount of temporary data is required. Though we currently do not this, we could also that handle transparency by storing 3 floats per ray for the opacity value, and disabling rays if this opacity crosses a given threshold.

The main difference compared to primary frusta is that the *delta* vector is no longer created for a fixed 4×4 pixel grid but for a set of random samples on the light source plane. As (4×4) interleaved sampling [Keller and Heidrich 2001] requires a different set of random numbers for each of the 16 pixels, 16 *delta* vectors are created. As these samples are constant over the frame, the 16 *delta* vectors can be pre-generated per frame. Instead of using the four corners of the light source to generate the corner rays and frusta, it is slightly more beneficial to compute tighter bounds based on the axis-aligned bounds (in 2D) of the random samples set.

4.5 Shading, Interleaved Sampling and Filtering

Every sample on the light has different influence on the illumination at a given point. As shading every shadow ray would be quite expensive, we follow the same approximation as other soft shadow algorithms: the separation of the visibility term and the shader evaluation. A single sample to the center of the light is used for shading, while the fractional visibility is done by *MFT* (see Figure 3 for an illustration of the achieved soft shadow quality).

After determining primary visibility, each pixel is assigned to a different interleaved sampling set, using a 4×4 pixel grid pattern. Each set contains 16 2D random numbers, referring to 16 3D points on the light source plane (used to generate the shadow rays on demand). Once combined, all 2D sets yield 256 2D random numbers. In order to achieve high quality results, we use either Hammersley or Larcher-Pillichshammer QMC samples [Kollig and Keller 2002].

After tracing all shadow ray frusta, we perform a simple irradiance filtering pass over the image, filtering the illumination in a 4×4 neighborhood around every pixel. Each pixel needs therefore additional information such as normal, depth, and irradiance, which can be stored in the reduced 16-bit floating point format. The filtering itself is performed for 16 pixels in parallel. Even for our non-optimized implementation, the filtering itself requires less than 20 million cycles (on one core). Compared to tracing soft shadow frusta, these costs are rather negligible.

5 Results

A fair comparison between *MFT* and packet tracing requires that optimal settings are applied to both. A deep BVH in particular is sub-optimal for *MFT*, as the frusta typically prohibit efficient culling deep down in the tree. In contrast, packet tracing relies on efficient culling by the BVH to reduce the number of triangle intersection tests to a tolerable level. A deep BVH is therefore used for packet tracing and a shallower BVH for *MFT*. The shallower BVH would also lead to faster build/update times, but this is not considered in our comparison. The reference packet implementation uses an optimized BVH packet traversal with a Moeller-Trumbore triangle test [Möller and Trumbore 1997].

5.1 Comparison to 16-ray packet tracing

As the code for a ray packet-AABB intersection test has the same complexity as a ray interval-AABB intersection test (except for the zero-interval-fix), we will refer to them as just *AABB* tests.

Even assuming perfect coherence (which is not the case), traversing 16 ray packets by multi-frusta traversal would theoretically allow for reducing the number of AABB test (*#AABB*) by at most $16\times$. However, due to the slightly shallower BVH tree the reduction in *#AABB* tests is actually slightly higher than $16\times$ (see Table 1): For the T-Rex scene, multi-frusta traversal in combination with a shallow BVH reduces *#AABB* tests by up to factor of $21\times$ for primary rays *P-Frusta*, while for soft shadows with 16 samples per frustum (*S-Frusta*) this value increases to $23.8\times$. Statistics for soft shadows include primary ray results. The saving obviously vary depending on scene and the light source configuration, but in general *MFT* works even better for shadow rays than for primary rays.

As a single triangle is typically tested for intersection by multiple frusta, $1.58 - 3.67\times$ less triangles are accessed in total, saving memory bandwidth. Moreover, corner ray culling compensates the pure frusta-based traversal by reducing the number of triangles passed to the 16-wide ray-triangle intersections tests (*#tris* + culling) by $2.98 - 10.06\times$. At the same time, the culling efficiency reduces the actual number of 16-wide triangle intersection test (*#isec*) by $1.01 - 3.1\times$.

As multi-frusta traversal has higher setup cost than packet tracing and corner ray culling adds additional cost, the speedup in run-time performance ranges from $3.42\times$ to $7.83\times$ for primary rays without shading. Applying even simple shading decreases the speedup factor to $2.98 - 6.09\times$. For tracing primary rays, shading, filtering and soft shadows, the speedup factor increases again to $3.34 - 6.11\times$.

	#tris (+culling)	#AABB tests	#isec tests	Simulation cycles
T-Rex (69K triangles)				
P-Packet (+shad)	399K	1218K	399K	94.1M (104.3M)
P-Frusta (+shad)	173K / 42K	58K	125K	19.5M (29.6M)
Reduction	**2.3x / 9.5x**	**21x**	**3.1x**	**4.82x (3.52x)**
S-Packet	10957K / 7298K	36234K	7698K	2572M
S-Frusta	3575K / 725K	1518K	5527K	635M
Reduction	**3.06x / 10.06x**	**23.8x**	**1.39x**	**4.05x**
Fern (212K triangles)				
P-Packet (+shad)	655K / 466K	2408K	466K	163.9M (175.2M)
P-Frusta (+shad)	414K / 156K	204K	315K	47.9M (58.6M)
Reduction	**1.58x / 2.98x**	**11.8x**	**2.08x**	**3.42x (2.98x)**
S-Packet	15882K / 8277K	34197K	10801K	3272M
S-Frusta	4325K / 1668K	2414K	10606K	977.5M
Reduction	**3.67x / 4.96x**	**14.1x**	**1.01x**	**3.34x**
Call of Juarez: Saloon (60K triangles)				
P-Packet (+shad)	383K / 278K	5839K	278K	246.9M (257.4M)
P-Frusta (+shad)	204K / 34K	388K	150K	31.5M (42.2M)
Reduction	**1.86x / 8.17x**	**15.04x**	**1.85x**	**7.83x (6.09x)**
S-Packet	5332K / 3209K	81023K	3209K	2672M
S-Frusta	2730K / 340K	5419K	2218K	437M
Reduction	**1.95x / 9.43x**	**14.95x**	**1.44x**	**6.11x**
Fairy Forest (174K triangles)				
P-Packet (+shad)	587K / 524K	3651K	524K	209M (222.5M)
P-Frusta (+shad)	403K / 111K	235K	312K	49.5M (63.2M)
Reduction	**1.45x / 4.61x**	**15.5x**	**1.67x**	**4.2x (3.52x)**
S-Packet	6556K / 4290K	51765K	4290K	2535M
S-Frusta	3461K / 775K	3121K	3874K	637M
Reduction	**1.89x / 5.53x**	**16.58x**	**1.1x**	**3.97x**

Table 1: Statistical and run-time performance (1024×1024 resolution, single simulated 1 GHz LRB core) comparison between *MFT* and packet tracing. Multi-frusta traversal for primary rays (*P-Frusta*) and for soft shadows with 16 samples per frustum (*S-Frusta*) provides a reduction of $11.8-23.8\times$ over standard packet traversal (*P-Packet* and *S-Packet*) in the number of ray-AABB intersection tests (#AABB tests). In addition, back-face and corner ray culling (*#tris + culling*) significantly reduce the number of triangles passed to the intersection test. For primary rays, a run-time speedup of $3.42-7.83\times$ ($2.98-6.09\times$ including shading) is achieved. For soft shadows with 16 samples per frustum, including primary rays, shading, and filtering, *MFT* provides a speedup of $3.34-6.11\times$.

5.2 Comparison to 256-ray packet tracing

We also compared the performance of *MFT* to an optimized packet-tracing implementation for $16 \times 16 = 256$ rays. To achieve a fair comparison, only the traversal of *MFT* has been replaced, while the shallow BVH, corner culling, and triangle intersection tests have been left in place. For each BVH node, we iterate over all active 16-wide ray packets and determine whether it intersects the children AABBs. This results in two new active masks for the left and right child. At the leaf, the current active mask is used in the same way as *MFT* does.

Even though the statistical results are slightly better than for *MFT* (3-4% less #AABB tests on average), 256-ray packet tracing reaches at most 50% of the performance of *MFT*. In some cases, the run-time performance is actually slower than 16-wide packet tracing. The main reasons for the bad performance are the increased memory and computation cost (ray directions and reciprocals), and far more complex traversal code. In case only a few 16-wide packets are active, the complex traversal slows down performance.

5.3 Frustum shrinking

As *MFT* shifts the cost from traversal to intersection (in particular for soft shadows), it is worth exploreing techniques for reducing the number of intersection tests further. Using the techniques described so far, we can terminate individual shadow frusta once all the frustum's rays are occluded, but in particular in penubra region may still end up with frusta that are partially occluded.

	#tris (+culling) accesses	#AABB tests	#isec tests	# addl. tests
T-Rex (69K triangles), 1024×1024				
No Shrink.	3401K/613K	1459K	5402K	
Ray-AABB	2896/498K	3733K	3973K	
	-14.8%/-18.6%	**155%**	**-26.4%**	
CR	3401K/512K	1459K	4261K	386K
	0%/-13.9%	**0%**	**-21.1%**	
FS	3138K/503K	1433K	4168K	386K
	-7.7%/-17.9%	**-1.7%**	**-22.8%**	

Table 2: Three techniques to implicitly or explicitly shrink frusta. *Ray-AABB* performs intersection tests between the leaf's AABB and all non-terminated shadow ray packets, resulting in a lot of additional intersection tests. *CR* only shrinks the extend of the corner rays on the light source plane, while *FS* shrinks the ray direction interval bounds too. For our implementation only *CR* is a viable option, as it has the best triangle reduction vs. overhead ratio.

For such partially occluded packets, *shrinking* the bounding frustum such that the frustum bounds only active rays and ignores inactive ones would lead to tighter frusta and, consequently, more culling and fewer traversal steps. There are multiple different ways of how this shrinking can be done, and these vary in both effectiveness and cost overhead. We will investigate three different options (also see Figure 2).

Ray packet-AABB intersection tests at the leaves For each half-occluded frustum, an additional ray-packet intersection test with the leaf's AABB is performed. Although this test reduces the number of triangle intersection tests by 26.4%, it leads to a significant increase in additional AABB tests: while a typical traversal step performs one SIMD AABB test for all 16 frusta in parallel, leaf culling has to perform a SIMD AABB test for every active frustum, leading to an increase in AABB tests by +155%. In addition, these per-frustum AABB tests are more costly, since they require per-ray direction reciprocals, which are costly to compute. Overall, leaf culling does not pay off, since its overhead is higher than the 26.4% savings in triangle tests.

Recomputing the corner rays A much simpler version of shrinking a frustum is to only compute the 2D parameter interval (on the camera/light plane) of the still-active samples, and recompute the corner rays through the corners of thsi 2D rectangle. Partially occluded frusta then have tighter corner rays, and cull more triangles. At a culling rate of 21.1%, this method is almost as effective as performing ray packet-AABB tests (26.4%), but has far lower cost that actually pays off (see Table 2). We currently perform this shrinking after all triangles in a leaf have been intersected, but only if at least one shadow ray has newly terminated.

Shrinking the frustum intervals Recomputing the corner rays leads to better triangle culling, but does not affect which nodes are traversed, and thus, does not reduce the number of accessed triangles. To have an impact on traversal steps, one has to shrink the interval arithmetic bounds used by the BVH traversal. Doing so every time the corner rays are recomputed indeed reduces the number of accessed triangles by an additional 7.7%. However, the achieved reduction is marginal, and does not pay for the additional cost in recomputing the tightest frusta and associated direction reciprocals.

In summary, for both leaf-culling and frustum interval shrinking the statistical gains in reduced traversal and intersection steps were outweighed by the respective method's overhead. The only successful option for our implementation was to shrink the corner rays, but even that led to a maximum speedup of only 10%.

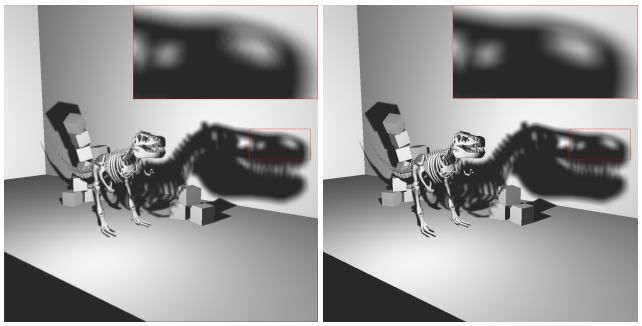

Figure 5: Left: MFT with interleaved sampling and discontinuity filtering, requiring only 16 shadow rays per pixel, Right: Packet tracing with 256 shadow rays per pixel.

5.4 Shadow Quality and Impact of LS Size

Even for complex shadow patterns like the T-Rex scene, $4 \times 4 = 16$ interleaved sampling in combination with a 4×4 pixel discontinuity filtering achieves almost the same quality as shooting 256 shadow rays per pixel, but requires only a $\frac{1}{16}$ in the number of shadow rays, see Figure 5.

If improved quality is required, one could either use directly more sample sets per frustum or perform a second pass over the image and adaptively refine only the penumbra pixels. As the number of penumbra pixels is typically only a fraction of all pixels, the second approach would be quite efficient.

As shown in Figure 6, the total number of accessed triangles (without culling) and triangle intersections tests increases linearly with the light source size.

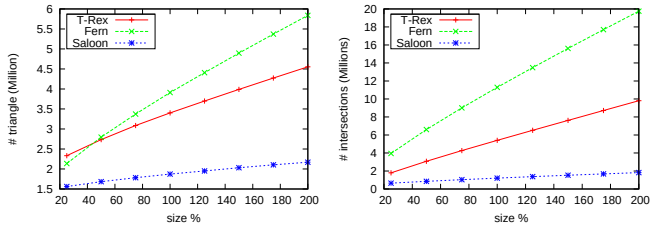

Figure 6: Impact of light source size for the T-Rex, Fern, and Saloon scene. Left: Total number of accessed triangles without culling. Right: Total number of triangles intersection tests. Both increase linearly with the light source size.

6 Discussion

Like another packet or frustum based traversal technique, the efficiency of MFT depends significantly on the kind—and coherence—of the rays that it is being applied to. Traditional packet tracing can, at worst, lead to total loss of SIMD efficiency and a deterioration to single-ray traversal. In contrast to this, like other large-packet and frustum traversal techniques, MFT can theoretically even lead to even worse perforamnce when applied to incoherent rays that span excessively wide frusta (see, e.g. [Wald et al. 2007; Wald et al. 2006]). For all reasonably coherent ray distributions, such as primary, hard, and soft shadow rays, however, it outperforms packet tracing. For all incoherent ray distributions, the technique does not perform well, as the frusta become too wide to provide efficient culling. In these cases, it would be useful to switch to another traversal algorithm which is better suited for incoherent rays but still benefits from wide SIMD [Wald et al. 2008a; Boulos et al. 2008; **?**; **?**].

The requirement that a pre-built BVH is available is not a hard limitation as typical ray tracing systems either rebuild or refit their BVH per frame to handle dynamic scenes. Recent work has shown [Wald 2007; Wald et al. 2008b; Lauterbach et al. 2009], that building BVH can be efficiently mapped to a parallel architecture and is fast enough to fully rebuild the BVH for moderately complex scenes. Obviously, lazy building can be used for *MFT* too, further lowering the build impact. As the built BVH is view-independent, as opposed to [Hunt and Mark 2008], it can be reused for all light sources, amortizing the build cost further.

Assuming—like [Seiler et al. 2008]—a LRB core with a hypothetical clock rate of 1 GHz, our method would achieve over 30 million rays per second per core. Similar to other ray tracing systems on Larrabee [Seiler et al. 2008], our approach scales linearly with the number of cores, so a hypothetical number of 16 such cores would achieve over half a billion rays per second, and 16-36 frames per second at a resolution of 1024×1024. Compared to soft shadow algorithms based on rasterization using e.g. shadow wedges, shadow volumes, soft irregular Z-Buffer, etc., the performance of our approach lies within $2\times$ of what has been reported in [Johnson et al. 2009]. However, all these approaches use specialized data structures and other approximations, which complicates an easy integration into an existing ray-tracing system.

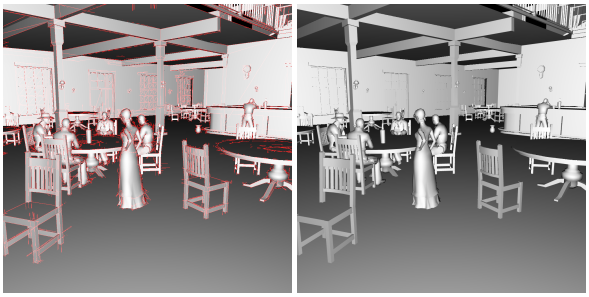

Figure 7: *MFT* allows for a fast and easy integration of adaptive sampling, in particular for MSAA or super-sampling. Left image: Silhouette edges are determined during traversal. For the Saloon scene, roughly 11.5% of all pixels are marked for retrace. Right image: Image quality when all silhouette marked pixels are retraced with 16 samples per pixel.

6.1 Other applications

While we have so far only considered accelerating primary and soft shadow rays, MFT can also be used for other applications, as long as ray density and coherence is sufficiently high. For example, for a packet-based instant global illumination based system that uses 4×4 interleaved sampling, one could directly connect each pattern's hit point to a different virtual point light (VPL) each, again forming 16 packets of 16 shadow rays each (in that case, the main difference would be that one traces the packets from the VPLs towards the surface rather than vice versa). The same framework can also be used to compute environment illumination (connecting 16 interleaved surface samples to the same envmap sample), for ambient occlusion, etc.

Alternatively, one can also use our technique for fast MSAA-like supersampling, by tracing 16 visibility samples per pixel to determine partial occlusion and/or silhouettes. Since each frustum would then represent an entire pixel, with minimum modifications MFT even allows for cheaply determining pixels containing silhouette edges. Once adjacency information is available, an edge can be identified as a silhouette edge by comparing the two adjacent triangles' orientation with respect to the frustum's origin (also see Figure 7).

7 Conclusion and Future Work

For coherent rays, *MFT* reduces the number of AABB intersection tests by up to $23\times$. All steps of our *MFT* technique map very well to Larrabee's wide SIMD architecture, and achieve a very high performance for primary rays, hard shadow rays and in particular for soft shadows. It outperforms packet tracing by a factor of $3-7\times$ for primary rays and by $4-6\times$ for soft shadows.

The approach is simple and easy to integrate into existing packet-based ray tracing engines. It does not require the traversal of additional data structures, nor significant changes to the rendering core itself. In particular, MFT can also be used within an exiting 16-ray packet traverser, and use it only where it makes sense (eg, primary or shadow rays), while relying on packet tracing for secondary rays.

Due to the low memory footprint it should map very well to today's GPU architectures. As *MFT* is a very general concept, it is possible to apply it to various applications where coherent ray distributions are given. The same *concept* of processing 16 *different* frusta in parallel would also make sense for other traversal algorithms like frustum-based kd-tree or octrees traversal, or coherent grid traversal [Wald et al. 2006].

In the future, we would like to explore hierarchical traversal for primary visibility and higher resolutions. This could be used to quickly find *entry points* in the BVH for larger than 16×16 pixel blocks, further reducing AABB intersection tests. Instead of handling individual triangles at the leaves, edge-based representations look promising. For improved soft shadow quality without taking additional samples, we would like to investigate larger and more advanced filter kernels, and the support for transparent shadows.

ACKNOWLEDGEMENTS

We thank Alexander Reshetov and Manfred Ernst for fruitful discussions and the anonymous reviewers for their valuable comments.

References

AMANATIDES, J. 1984. Ray tracing with cones. In *SIGGRAPH '84*, ACM, 129–135.

ANNEN, T., DONG, Z., MERTENS, T., BEKAERT, P., SEIDEL, H.-P., AND KAUTZ, J. 2008. Real-time, all-frequency shadows in dynamic scenes. *ACM Trans. Graph. 27*, 3, 1–8.

ASSARSSON, U., DOUGHERTY, M., MOUNIER, M., AND AKENINE-MÖLLER, T. 2003. An optimized soft shadow volume algorithm with real-time performance. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, 33–40.

BENTHIN, C. 2006. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University.

BOULOS, S., WALD, I., AND SHIRLEY, P. 2006. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Tech. Rep. UUCS-06-010, SCI Institute, University of Utah.

BOULOS, S., WALD, I., AND BENTHIN, C. 2008. Adaptive ray packet reordering. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*.

COOK, R., PORTER, T., AND CARPENTER, L. 1984. Distributed Ray Tracing. *Computer Graphics (Proceeding of SIGGRAPH 84) 18*, 3, 137–144.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

DRETTAKIS, G., AND FIUME, E. 1994. A fast shadow algorithm for area light sources using backprojection. In *SIGGRAPH 94 Proceedings*, ACM, 223–230.

FERNANDO, R. 2005. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, ACM, 35.

HART, D., DUTRÉ, P., AND GREENBERG, D. P. 1999. Direct illumination with lazy visibility evaluation. In *SIGGRAPH '99*, ACM, 147–154.

HASENFRATZ, J.-M., LAPIERRE, M., HOLZSCHUCH, N., AND SILLION, F. 2003. A survey of real-time soft shadows algorithms. *CG Forum 22*, 4, 753–774.

HUNT, W., AND MARK, W. R. 2008. Ray-specialized acceleration structures for ray tracing. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*.

INTEL LRBNI, 2009. C++ Larrabee Prototype Library. http://software.intel.com/en-us/articles/prototype-primitives-guide/.

JOHNSON, G. S., HUNT, W. A., HUX, A., MARK, W. R., BURNS, C. A., AND JUNKINS, S. 2009. Soft irregular shadow mapping: fast, high-quality, and robust soft shadows. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, 57–66.

KAY, T., AND KAJIYA, J. 1986. Ray tracing complex scenes. In *Proceedings of SIGGRAPH*, 269–278.

KELLER, A., AND HEIDRICH, W. 2001. Interleaved Sampling. *Proceedings of the 12th Eurographics Workshop on Rendering*, 269–276.

KELLER, A. 1997. Instant Radiosity. *Proceedings of ACM SIGGRAPH*, 49–56.

KOLLIG, T., AND KELLER, A. 2002. Efficient Multidimensional Sampling. *Computer Graphics Forum 21*, 3, 557–563. (Proceedings of Eurographics 2002).

LAINE, S., AILA, T., ASSARSSON, U., LEHTINEN, J., AND AKENINE-MÖLLER, T. 2005. Soft shadow volumes for ray tracing. *ACM Transactions on Graphics 24*, 3, 1156–1165.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE2, D., , AND MANOCHA, D. 2009. Fast bvh construction on gpus. In *Eurographics 2009: Proceedings*.

LEHTINEN, J., LAINE, S., AND AILA, T. 2006. An improved physically-based soft shadow volume algorithm. *Computer Graphics Forum 25*, 3.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools 2*, 1, 21–28.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2007. A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Proceedings of the Eurographics Symposium on Rendering*.

PARKER, S., SHIRLEY, P., AND SMITS, B. 1998. Single Sample Soft Shadows. Tech. Rep. UUCS-98-019, Computer Science Department, University of Utah, October. Available at http://www.cs.utah.edu/∼bes/papers/coneShadow.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics 24*, 3, 1176–1185. (Proceedings of ACM SIGGRAPH 2005).

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3, 1–15.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 527–536.

SOLER, C., AND SILLION, F. 1998. Fast calculation of soft shadow textures using convolution. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, 321–332.

STEWART, A. J., AND GHALI, S. 1994. Fast computation of shadow boundaries using spatial coherence and backprojections. In *SIGGRAPH 94 Proceedings*, ACM, 231–238.

VAN DER ZWAAN, M., REINHARD, E., AND JANSEN, F. 1995. Pyramid clipping for efficient ray traversal. In *Rendering Techniques '95, Proceedings of the Eurographics Workshop on Rendering*, 1–10.

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

WALD, I., KOLLIG, T., BENTHIN, C., KELLER, A., AND SLUSALLEK, P. 2002. Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques*, 15–24. (Proceedings of the 13th Eurographics Workshop on Rendering).

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM TOG 25*, 3, 485–493.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM TOG 26*, 1, 1–18.

WALD, I., BENTHIN, C., AND BOULOS, S. 2008. Getting rid of packets: Efficient simd single-ray traversal using multi-branching bvhs. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*.

WALD, I., IZE, T., AND PARKER, S. G. 2008. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics*.

WALD, I. 2007. On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing*, 33–40.

Appendix A

Pseudo C++ traversal code for primary rays assuming common origin and directions. All operations on vector classes can directly be implemented by the LRBni instruction prototype library.

```
lrb_f  /* vector SIMD class for 16 float values */
lrb2f  /* vector SIMD class for 2 x 16 float values */
lrb3f  /* vector SIMD class for 3 x 16 float values */
lrb_m  /* class for the 16-bit mask type */

struct StackNode {
  int node;
  lrb_m mask;
};

/* --- 3D slabs test for common origin rays --- */
/* --- using interval arithmetic           --- */
/
lrb2f SlabsTest(float *aabb,
                int offset[3],
                lrb3f min_rcp,
                lrb3f max_rcp,
                lrb_f distance)
{
  lrb_f clipMinX = aabb[offset[0]]*max_rcp[0];
  lrb_f clipMinY = aabb[offset[1]]*max_rcp[1];
  lrb_f clipMinZ = aabb[offset[2]]*max_rcp[2];
  lrb_f clipMaxX = aabb[offset[0]^4]*min_rcp[0];
  lrb_f clipMaxY = aabb[offset[1]^4]*min_rcp[1];
  lrb_f clipMaxZ = aabb[offset[2]^4]*min_rcp[2];
  lrb_f min_dist = max(max(clipMinX,clipMinY),clipMinZ);
  lrb_f max_dist = min(min(clipMaxX,clipMaxY),clipMaxZ);
  return lrb2f(max(min_dist,0.0f),min(max_dist,distance));
}

void TraceMultiFrusta(...) {
  /* --- origin stored in 4 x AOS layout [x y z 0] --- */
  lrb_f origin4;
  /* ---  16 distances, each entry is the maximum --- */
  /* --- distance of all rays per frustum        --- */
  lrb_f distance;

  /* --- init stack with root node and      --- */
  /* --- set all bits in frusta active mask --- */
  StackNode stack[MAX_STACK_SIZE];
  stack[0].node = 0;
  stack[0].node = 0xffff;
  int stackIndex = 1;

  /* --- reciprocal ray direction intervals --- */
  lrb3f min_rcp, max_rcp;

  /* --- offset table to select min/max --- */
  int offset[3];
  offset[0] = (lz(min_rcp[0]) == 0) ? 0 : 4;
  offset[1] = (lz(min_rcp[1]) == 0) ? 1 : 5;
  offset[2] = (lz(min_rcp[2]) == 0) ? 2 : 6;

  while (1) {
    pop_stack_entry:
    if (stackIndex == 0) break;
    stackIndex--;
    /* ---  current BVH node index and frusta active mask --- */
    int node = stackNode[stackIndex].node;
    lrb_m mask = stackNode[stackIndex].mask;
    while (1) {
      if (isLeaf(bvh[currentNode])) break;
      int childrenID = firstChildID(bvh[node]);

      /* --- subtract origin from left and right child AABB --- */
      lrb_f box16 = lrb_f((float*)&bvh[childrenID]) - origin4;
      float *leftAABB  = (float*)&box16;
      float *rightAABB = (float*)&box16 + 8;

      /* --- IA slabs test for left and right AABB --- */
      lBounds = SlabsTest(leftAABB,offset,min_rcp,max_rcp,distance);
      rBounds = SlabsTest(rightAABB,offset,min_rcp,max_rcp,distance);

      /* --- mask = bounds_min <= bounds_max --- */
      lrb_m leftMask = le(lBounds[0],lBounds[1]);
      lrb_m rightMask = le(rBounds[0],rBounds[1]);

      if (leftMask && rightMask) {
        /* --- decide order when traversing both children --- */
        if (lt(lBounds[0],rBounds[0])) {
          node = childrenID+0; mask = leftMask;
          stackNode[stackIndex].node = childrenID+1;
          stackNode[stackIndex].mask = rightMask;
          stackIndex++;
        } else {
          node = childrenID+1; mask = rightMask;
          stackNode[stackIndex].node = childrenID+0;
          stackNode[stackIndex].mask = leftMask;
          stackIndex++;
        } }
      else if (leftMask || rightMask) {
        /* --- traverse either left or right child --- */
        if ( leftMask ) {
          node = childrenID + 0; mask = leftMask;
        } else {
          node  = childrenID + 1; mask = rightMask;
        }
      }
      else goto pop_stack_entry;
    }

    /* --- leaf intersection --- */
    lrb_m updateMask = 0;
    for (int i=0;i<triangles;i++) {
      /* --- for each triangle perform back-face --- */
      /* --- and corner ray culling to create    --- */
      /* --- the 'triangleActiveMask'            --- */

      lrb_m triangleActiveMask = mask & ...

      /* --- iterate over all 'active' frusta ---*/
      int frustumID = -1;
      while((frustumID = bitscan(frustumID,triangleActiveMask)) != -1)
      {
        /* --- generate rays on demand, perform triangle --- */
        /* --- intersection, and initialize 'updateMask' --- */
        ...
        updateMask |= 1 << frustumID;
      }
    }

    /* --- update 'distance' but only for those --- */
    /* --- frusta marked by the 'updateMask'    --- */
    int frustumID = -1;
    while((frustumID = bitscan(frustumID,updateMask)) != -1)
    {
      distance[frustumID] = ...
    }
  }
}
```