Embree: A Kernel Framework for Efficient CPU Ray Tracing

Ingo Wald [⋆†] Sven Woop [⋆†] Carsten Benthin [⋆†] Gregory S. Johnson [⋆†] Manfred Ernst [⋄‡]

[⋆] Intel Corporation [⋄] now at Google Incorporated

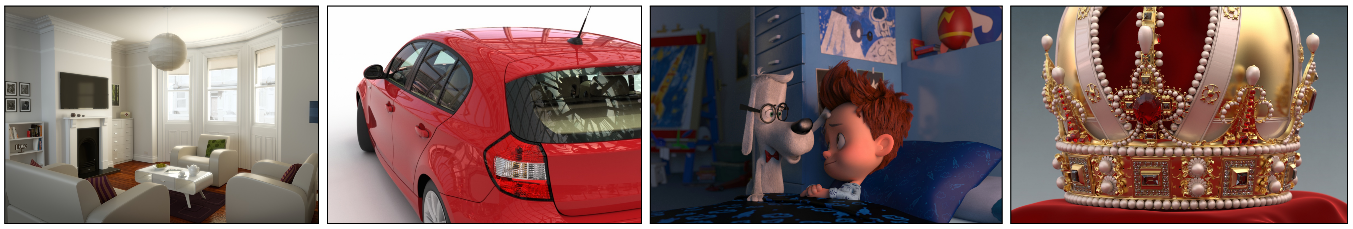

Figure 1: *Images produced by renderers which use the open source Embree ray tracing kernels. These scenes are computationally challenging due to complex geometry and spatially incoherent secondary rays. From left to right: The White Room model by Jay Hardy rendered in Autodesk RapidRT, a car model rendered in the Embree path tracer, a scene from the DreamWorks Animation movie "Peabody & Sherman" rendered with a prototype path tracer, and the Imperial Crown of Austria model by Martin Lubich rendered in the Embree path tracer.*

Abstract

We describe Embree, an open source ray tracing framework for x86 CPUs. Embree is explicitly designed to achieve high performance in professional rendering environments in which complex geometry and incoherent ray distributions are common. Embree consists of a set of low-level kernels that maximize utilization of modern CPU architectures, and an API which enables these kernels to be used in existing renderers with minimal programmer effort. In this paper, we describe the design goals and software architecture of Embree, and show that for secondary rays in particular, the performance of Embree is competitive with (and often higher than) existing state-of-the-art methods on CPUs and GPUs.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—[Ray Tracing]

Keywords: ray tracing, SIMD, SPMD, CPU, coprocessor

Links: DL PDF WEB

1 Introduction

The combination of Moore's law and the rapid growth in functional unit replication has substantially increased the compute capability in modern CPUs. However, efficiently exploiting this capability for ray tracing is challenging. For example, kernels for stepping a ray through a spatial data structure frequently exhibit fine-grained data dependent branching and irregular memory access patterns, inhibiting auto-vectorization. Worse, the optimal mapping of a kernel to a specific architecture is not always obvious, even for an experienced programmer. Application and workload-specific constraints can also affect the choice of data structure and traversal algorithm, which in turn may affect the vectorization strategy. Consequently, many ray tracing applications do not use the most efficient combination of algorithms, data structures, and parallelization strategy for the target architecture, and so run slower than they could.

Embree is an open source ray tracing framework that enables high performance in new and existing renderers by efficiently exploiting the full compute capability of modern x86 architectures. Embree is directed at professional rendering environments in which detailed geometry and secondary illumination effects with incoherent ray distributions are common. To achieve high performance across a wide range of workloads and x86 architectures, Embree includes a suite of hand optimized, low-level kernels for accelerating spatial data structure construction and traversal. For example, both packet and single ray intersection queries are supported as well as logic for dynamically switching between these methods at runtime. These kernels are accessed through a straightforward and flexible API that minimally constrains the design of the calling application.

2 Related Work

Recent work in ray tracing has largely focused on minimizing the total number of rays needed to render a high fidelity image, or on improving the average performance of traced rays [Glassner 1989; Havran 2000; Wald 2004]. Embree addresses the latter, primarily by providing hand-vectorized kernels for x86 CPUs which support the SSE and AVX instruction sets.

2.1 Packet Tracing

It is possible to vectorize ray traversal of spatial data structures such as bounding volume hierarchies (BVH) by assigning rays to vector lanes, and tracing this ray "packet" through the BVH. Since packet tracing is independent of the vector unit width or instruction set architecture (ISA), it is broadly applicable and can achieve high vector utilization for spatially coherent rays. However, packet tracing performs less well for incoherent rays, and the renderer itself

[†] ingo.wald, sven.woop, carsten.benthin, gregory.s.johnson@intel.com
[‡] ernst.manfred@gmx.de

http://doi.acm.org/10.1145/2601097.2601199

must be parallelized so that multiple rays are available to be traced together. This dependence on spatial coherence and parallelism in the renderer inhibits the use of packet tracing.

Though classical packet tracing traces a single ray per vector lane, it is also possible to trace packets with (effectively) more rays than lanes. For example, multi-frusta methods trace a frustum per lane, and each frustum is a proxy for a set of rays bounded by the frustum volume [Reshetov et al. 2005; Benthin and Wald 2009]. These techniques can in some cases achieve higher performance relative to simple packet tracing, but are even more sensitive to ray coherence.

2.2 Single-Ray Vectorization

Other work has focused on using vectorization to accelerate spatial data structure traversal for individual rays. For example, multi-branching BVH structures enable multiple nodes or primitives to be tested for intersection against the same ray in parallel. Quad-branching BVH (BVH4) data structures perform well with 4-wide and 16-wide vector units [Ernst and Greiner 2008; Dammertz et al. 2008; Benthin et al. 2012], but higher branching factors offer diminishing returns [Wald et al. 2008].

Generally speaking, single ray vectorization is faster than packet tracing for incoherent ray distributions (and can be used within a scalar renderer), but is slower for coherent rays. For this reason, hybrid techniques have been developed which can dynamically switch between packet tracing where rays are coherent, and single-ray vectorization where not [Benthin et al. 2012].

2.3 Ray Tracing Systems

Several high performance ray tracing systems exist for both CPUs and GPUs [Wald et al. 2002; Bigler et al. 2006]. OptiX is a state-of-the-art system designed to simplify development of new renderers for GPUs [Parker et al. 2010]. OptiX consists of low level kernels, a programmable ray tracing pipeline, a domain specific programming model and compiler, and a scene graph specification. Much of this infrastructure is needed to transparently split work between the CPU and GPU, schedule work items across cores, and page data between memory on the host and GPU. In contrast, a key design goal for Embree is to accelerate ray tracing in *existing* renderers for CPUs, with maximal flexibility and minimal programmer effort. This design goal leads to a low-level kernel framework rather than a complete system.

3 Embree Design Goals

The development of Embree is guided by four observations. First, even full-featured photorealistic renderers can be built from a small set of common ray tracing operations (e.g. spatial data structure build and traversal). Second, CPU architectures are in principle suited to compute intensive workloads with abundant fine-grained data dependent branching (e.g. hierarchical data structure traversal), but achieving high throughput is challenging in practice. Third, ray tracing is widely used in professional rendering applications, and there is a need to achieve high performance in these applications. Fourth, much recent work in accelerating ray tracing has not found its way into actual use due to the complexity of integrating these techniques into existing renderers (e.g. parallel data structure build, single-ray vectorization) or because the techniques are highly specialized (e.g. packet tracing). Three design goals follow from these observations.

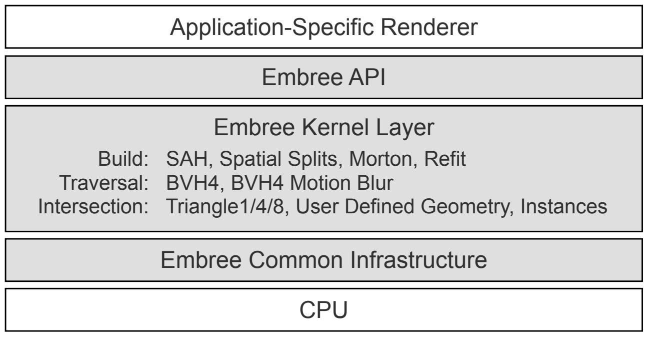
Application-Specific Renderer
Embree API
Embree Kernel Layer
Build: SAH, Spatial Splits, Morton, Refit
Traversal: BVH4, BVH4 Motion Blur
Intersection: Triangle1/4/8, User Defined Geometry, Instances
Embree Common Infrastructure
CPU

Figure 2: *The major components of a complete system which uses Embree. An application-specific renderer typically uses the Embree kernels through a compact API. The kernels implement the functionality required to build a spatial data structure and trace rays. The Embree common infrastructure layer provides cross-platform wrappers for low-level atomics, synchronization, threading, etc.*

3.1 Focus on Professional Rendering

Embree is intended to enable high performance ray tracing for professional rendering workloads. Two important characteristics of these workloads is that they include large models, and contain a mix of primary rays with high spatial coherence and secondary rays with low spatial coherence. For this reason, the Embree kernels are designed to exploit parallelism at multiple granularities, with an emphasis on efficient vectorization and architecture-specific, hand-optimized implementations.

3.2 Broad Applicability

Achieving maximal performance on modern processors requires exploiting parallelism in all layers of a rendering system. However, for several practical reasons (e.g. large existing rendering codebases with proprietary effects), a completely new rendering system is less likely to find wide adoption than a low-level kernel framework. As a result, Embree is focused on providing efficient parallel implementations of common ray tracing operations through a small set of kernels. This focus avoids placing restrictions on the design of the renderer, increasing the cases in which the kernels can be used.

3.3 Ease of Use

Embree is designed to be used in existing renderers with minimal effort. The ray tracing operations supported in Embree are exposed in a straightforward way via a compact and stable API. This API minimizes changes to existing render codes, and speeds adoption of future improvements in ray tracing algorithms supported in Embree (users need only download the latest version). In addition, Embree is fully open source to avoid licensing restrictions and allow users to inspect and optionally modify every part of its implementation.

4 Embree Overview

Figure 2 illustrates the major components of a complete rendering system which uses Embree. Though we provide a fully functional path tracer built on the Embree kernels (Section 7), a more common use case is to modify an existing application-specific renderer to use the kernels directly. The Embree kernels implement data structure construction and traversal, and ray-primitive intersection in each of several supported ISAs (Section 5). The kernels can be accessed through a straightforward API that hides implementation details such as data structure memory layout (Section 6). The kernels are in turn implemented over a common set of low-level components which include cross-platform wrappers for atomic and vector parallel operations, synchronization, and threading. Through this layer Embree supports all current x86 architectures, operating systems (Linux, Microsoft Windows, Apple Mac OS), and compilers (Intel C++ Compiler, GCC, Clang, and Microsoft Visual Studio).

4.1 Bounding Volume Hierarchy Variants

The Embree kernels operate exclusively on bounding volume hierarchies. Broadly speaking, BVH data structures have comparatively small memory footprints and low build times, and their generally shallow depth enables fast traversal. Further, the ability to fix the number of children per node facilitates single-ray SIMD traversal and intersection. We have found that a BVH branching factor of 4 is efficient for both packet and single-ray SIMD operations across multiple ISAs and workloads, relative to higher or lower branching factors (Subsection 2.2).

Embree supports BVH variants optimized for memory consumption or performance, to fit application-specific needs. The leaves of the `bvh4.triangle4i` subtype store references to primitives. This BVH is compact in memory, but traversal requires address indirection that inhibits prefetching, and costly memory gather operations that reduce the efficiency of intersecting primitives in parallel. In contrast, a leaf of the `bvh4.triangle4` subtype stores a pointer to a contiguous memory region containing the primitive data for that node. Ray traversal and intersection with this BVH subtype is significantly faster than in the former case, but requires more memory.

Embree also provides a BVH with support for linear motion blur [Grünschloßet al. 2011]. This subtype (`bvh4mb`) stores two sets of vertex coordinates, corresponding to the positions at times $t = 0$ and $t = 1$, from which the vertex position at time t is linearly interpolated during intersection. Similarly, two sets of bounds are stored per interior node, one for each value of t, and these bounds are interpolated during traversal. The interpolated bounds are not guaranteed to be optimal, but are conservatively correct.

4.2 Geometric Primitive Representations

Embree supports multiple primitive types. Each type is designed to maximize performance on a given ISA, reduce memory usage, or is optimized for an application-specific priority (e.g. consistency during intersection). A `triangle{1,4,8}{i,v,n}` primitive is a record that stores vertex indices (`i`), vertex data (`v`), or preprocessed edge and normal data (`n`) for 1, 4, or 8 triangles in a SIMD friendly memory layout. Each subtype implements an intersection method, which tests a single ray against the 1, 4, or 8 triangles. For example, the `triangleXn` subtypes implement Möller Trumbore intersection for performance [Kensler and Shirley 2006], and `triangle1n` is used during packet traversal (Subsection 5.1.3), while `triangle4n` and `8n` are used during single-ray and hybrid traversal on SSE and AVX respectively (Subsections 5.1.1, 5.1.4).

5 Embree Kernel Layer

The core component of Embree is a set of high performance kernels for constructing bounding volume hierarchies (Subsection 5.2) and tracing rays (Subsection 5.1). The tracing kernels are implemented internally using separate components for BVH traversal and ray-triangle intersection. Individual kernels can be composited, allowing construction and traversal of multi-level BVHs, data structures that span multiple meshes, and geometry instantiation (Section 6).

5.1 Traversal and Intersection Kernels

In principle, the number of traversal and intersection kernels needed to address the BVH and primitive types, vectorization methods, and ISAs supported by Embree is large. However, not all combinations yield high performance or are valid across all ISAs. Further, the Embree kernels are templated, enabling a single implementation to be used with multiple BVH and primitive representations. Lastly, conditional compilation is used to support specialized features such as face culling and ray masks.

5.1.1 Single-Ray Triangle Intersection

There are several ways a single ray can be tested for intersection in parallel against the triangle(s) in a BVH leaf node, and the approach used in Embree varies by ISA. The ray can be tested against multiple triangles in parallel (denoted by the primitive type `triangleXn` where $\texttt{X} > 1$), or the separate channels (e.g. x, y, z) of a single ray-triangle intersection test can be computed in parallel ($\texttt{X} = 1$), or a combination can be used ($\texttt{N} > \texttt{X} > 1$ where `N` is the vector width of the ISA). The choice of methods used in Embree on a given ISA is informed by a comparative analysis of the performance of the vectorization options suited to that ISA.

On processors with 4-wide SSE instructions, Embree supports both `triangle1n` and `triangle4n` primitive types, where the latter is generally faster. On processors with 8-wide AVX / AVX2 instructions, `triangle4n` and `triangle8n` types are supported. The two perform similarly on SandyBridge[1] CPUs, while `8n` is often faster (3%) on Haswell[2] CPUs due to L1 cache bandwidth optimizations. On machines equipped with Intel® Xeon Phi™ coprocessors, 16-wide IMCI instructions are available. Here, Embree operates on 4 triangles in parallel by treating the 16-wide vector units as a set of 4-wide vector units, each of which is assigned a `triangle1n` primitive [Benthin et al. 2012]. A `triangle16n` variant was tested, but found to inhibit construction of high quality BVHs, and is unused.

5.1.2 Single-Ray BVH Traversal

Quad-BVH structures in which child nodes are stored together in memory, enable efficient single-ray traversal on architectures with a vector width of 4 [Ernst and Greiner 2008; Dammertz et al. 2008]. At each traversal step, the ray is tested for intersection against the 4 child nodes in parallel.

The Embree single-ray BVH traversal kernels provide highly tuned implementations of this approach for several current ISAs. On SSE 4.1, higher throughput (around 3%) is achieved by mapping some floating point vector operations to integer units. This is possible for rays starting at or behind the origin, since a signed integer compare gives the correct result (and negative values remain smaller than positive values) when a floating point number is reinterpreted as an integer value. On AVX / AVX2, the same traversal method is used (we have found no way to effectively map the 4 box tests to 8-wide SIMD), with incremental performance improvements coming from AVX2 instructions such as fused multiply add (4%). On the Intel Xeon Phi coprocessor, the 16-wide IMCI instruction set is exploited (particularly horizontal shuffle, low latency loads) to perform the 4 box intersection tests with the x, y, and z components computed in parallel [Benthin et al. 2012]. In addition, the Embree traversal kernels seek to fill the available issue slots on superscalar architectures by generating the right mix of instructions. This optimization is particularly important on the Intel Xeon Phi coprocessor, which lacks out-of-order execution.

[1] Intel® Xeon® E5-2690 processor.
[2] Intel® Core™ i7-4770 processor.

5.1.3 Packet Traversal and Intersection

Packet tracing is conceptually simple compared to single-ray SIMD traversal and intersection. In classical packet tracing, rays in a given packet are intersected with the same BVH node or triangle at each step of traversal. However, it is also possible to implement packet tracing following a single program multiple data (SPMD) programming model. Here, the rays of a packet are independently traversed through the BVH, and each ray is potentially tested against different BVH nodes or triangles [Aila and Laine 2009].

The Embree packet kernels implement classical rather than SPMD packet tracing. This approach simplifies the control flow, enables the use of load-and-broadcast memory operations in place of costly gathers, and amortizes scalar computation across SIMD lanes. The performance is further improved through prefetching, minimizing data dependencies, and architecture-specific optimizations. The latter include generating the optimal mix of instructions needed to fill issue slots, and efficiently exploiting instructions unique to the ISA. For this reason, kernels are implemented for each ISA even though packet tracing is logically independent of the ISA vector width.

5.1.4 Hybrid Traversal and Intersection

In scenes with a mix of coherent and incoherent rays, BVH traversal and intersection performance can benefit from dynamically switching between packet and single-ray kernels [Benthin et al. 2012]. Yet the memory storage order for a BVH optimized for packets may be suboptimal for single-ray methods (the converse may also be true). For example, the Embree packet intersection kernels test multiple rays against a single triangle while the single-ray kernels typically intersect a single ray with multiple triangles. The storage order of the triangles is different in both cases (e.g. `triangle1n` versus `triangle4n`), but we have found the difference in packet tracing performance to be minimal.

For this reason, we allow a given BVH type in Embree to be used with both single-ray and packet traversal and intersection kernels, enabling the implementation of a hybrid ray traversal mode. This mode begins traversal using packets, and dynamically switches to single-ray traversal when the number of active rays in a packet falls below a threshold [Benthin et al. 2012]. This mode can improve traversal performance by 50% compared to packets alone.

5.2 BVH Construction Kernels

Embree is directed at professional rendering environments in which static or dynamic scenes with hundreds of millions of primitives are common. As a result, BVH construction performance is important. Embree provides build kernels focused on producing higher quality BVH structures (Subsection 5.2.1) or on higher performance BVH structures (Subsection 5.2.2). This distinction between quality and performance is relative, since both kernels are multithreaded and vectorized, and can achieve high build rates (Section 8.1).

5.2.1 Binned SAH BVH Construction

A high quality BVH is one in which the average ray traversal cost is minimized. Embree constructs a BVH optimized for traversal cost using object partitioning with the surface area heuristic (SAH) [Wald 2007] and spatial splits [Stich et al. 2009]. During construction, both methods are tested per node and the spatial partitioning which yields the lowest estimated cost is used.

The Embree implementation of the binned SAH algorithm proceeds in 3 stages. Threads cooperatively bin and partition triangles from the same BVH node(s) while the triangle count is large ($>$ 256K). Individual threads bin and partition triangles from different nodes and add the resulting child nodes to a shared work queue, when the triangle count is less than 256K but greater than 4K. Finally, threads process different nodes and their children to completion when the triangle count is small ($<$ 4K). For the `triangleX` types where `X` $>$ 1, the SAH cost function is modified to reflect that `X` triangles can be intersected for the same cost as one. This encourages (but does not guarantee) splits containing multiples of `X` triangles.

On the Intel Xeon Phi coprocessor, the binned SAH algorithm is modified to exploit the additional task parallelism available on this architecture. All threads initially bin and partition triangles from the same node, until the number of children exceeds the core count. Each core then processes a different node, with threads working cooperatively until the number of child nodes exceeds the thread count per core (4). Finally, different nodes are processed per thread. This modification improves overall build performance by up to 10%.

The Embree binned SAH kernel can be used with or without spatial splits for improved build quality or performance respectively. We use a modified form of spatial splits in which a single plane is tested at the spatial center of the node in each dimension, rather than multiple candidate planes along each axis. This method reduces build quality compared to the original algorithm but is faster and retains the key advantage – the ability to break up long diagonal triangles. Since spatial splits increase the number of triangle references, we place a global limit on scene size. Spatial splits are disabled once this limit is reached.

5.2.2 Morton-Code BVH Construction

The highest performance BVH build kernel in Embree uses Morton codes to express the construction process in terms of a simple radix sort [Lauterbach et al. 2009]. In the first stage of the algorithm, all threads cooperatively compute the centroid of the bounding box of each triangle, and the associated Morton code. Key / value pairs representing the code and associated triangle, are stored together in a single 64-bit value, and these pairs are sorted using a parallel radix sort. On the Intel Xeon Phi coprocessor, this implementation sorts 800 million pairs per second. The BVH hierarchy is then computed by partitioning triangles based on their Morton codes. Initially, all threads cooperatively partition triangles from the same node until the number of child nodes is larger than the number of threads, at which point different nodes are processed on different threads. The process terminates at a leaf when there are fewer than 4 triangles.

6 Embree API

The Embree kernels can be accessed through an included API. This API is straightforward, compact, and stable across Embree releases. The API includes functions for defining geometry, building a BVH over this geometry, and issuing intersection and occlusion queries. Figure 3 illustrates an example in which the Embree API is used to intersect a ray with a scene containing a triangle mesh. Though use of the API is not required, it can simplify application development. Importantly, the API hides technical details such as which combination of kernels and data layout in memory give the best performance for a given combination of geometry and ISA. The API assumes the kernels and application are in the same address space, and does not support an offload model for coprocessors.

6.1 Specifying Geometry

The Embree API supports several geometry types including triangle meshes, instances, and user-specified data types. A triangle mesh is defined by vertex data and the vertex indices composing each face. This is only the data necessary for BVH construction and primitive intersection. All other per-face or per-vertex data (e.g. shading normals) is considered to be in the domain of the application, ensuring that Embree remains compact and without the need to support more general data in a way that works with all possible applications.

User-defined geometry types are specified by providing bounding boxes and callback routines for ray-primitive intersection. This flexibility is similar to that provided in OptiX, although Embree supports custom callbacks via function pointers while OptiX uses a compiler framework to combine user-specified intersection kernels with OptiX traversal kernels.

6.2 Kernel Selection

Several options can be set on geometry to denote properties such as static or dynamic content, or motion blur. Other options are used to indicate application-specific priorities (e.g. BVH optimization for performance or memory usage), the anticipated ray distribution (e.g. coherent or incoherent), and the number of rays to be issued per query (e.g. 1, 4, 8, or 16). Embree chooses the combination of BVH construction and ray traversal kernels which are expected to yield the best performance for the given options and the current CPU. Kernels are selected at run-time via dynamic code dispatch.

6.3 BVH Configuration

Several configuration options affect the BVH type built for a given scene. For static scenes, a single `bvh4` is built over all geometry, with the primitive layout in memory (Subsection 4.2) chosen based on the BVH compactness and ray coherence options set by the user. For dynamic scenes, a two-level BVH is built with a separate `bvh4` per mesh [Wald et al. 2003]. Each sub-BVH is built using a binned SAH kernel in the case of static meshes, a refitting kernel in the case of deformable meshes, or a Morton-code builder in the case of fully dynamic geometry. As a special optimization for scenes with overlapping meshes, BVH nodes with a large surface area are iteratively replaced with their children until a threshold is reached.

7 Embree Sample Path Tracer

To illustrate the performance of the Embree kernels in a complete system, we have developed a fully functional path tracer with scalar and vectorized variations. The scalar renderer is used as a baseline for performance analysis, while the vectorized form indicates the performance potentially achievable in a fully parallel professional rendering application.

7.1 Scalar Reference Path Tracer

Our scalar path tracer is implemented in C++ using several common language features including templates, virtual functions, and STL containers. This renderer contains no explicit vectorization code, but does use the Embree Common Infrastructure (Figure 2), which defines several high-level tuple types (e.g. points, colors) and automatically and transparently maps these tuples and associated arithmetic operations to low-level SIMD types and intrinsics. Using this layer, it is possible to implement renderers in code which appears to be scalar, but which benefits to a degree from vectorization.

The design of this path tracer broadly follows that of PBRT [Pharr and Humphreys 2004], and includes the renderer itself, integrators, samplers, materials, and BRDFs. Ambient, directional, and point lights are supported, as well as an HDRI environment light with 2D importance sampling. Materials can be composed from one or more BRDFs (e.g. dielectric layers, microfacets), and several non-trivial materials are included like brushed metal and multi-layer metallic paint.

```
// create a container for scene geometry
RTCScene scene = rtcNewScene(...);

// add a triangle mesh object to the scene
unsigned id = rtcNewTriangleMesh(scene, ...);

// write vertex positions into vtx[] array
vtx = rtcMapBuffer(scene, id, RTC_VERTEX_BUFFER);
...
rtcUnmapBuffer(scene, id, RTC_VERTEX_BUFFER);

// write vertex indices per face into tri[] array
tri = rtcMapBuffer(scene, id, RTC_INDEX_BUFFER);
...
rtcUnmapBuffer(scene, id, RTC_INDEX_BUFFER);

// indicate that the scene is fully defined
rtcCommit(scene);

// initialize a ray
RTCRay ray;
...

// hit result is returned in ray.{geomID, ...}
rtcIntersect(scene, ray);

// discard the scene and its contents
rtcDeleteScene(scene);
```

Figure 3: *A simple example using the Embree API. A ray is traced into a scene with a single triangle mesh. A scene is a container for multiple geometries of potentially different types, while a geometry is a collection of primitives. This distinction allows Embree to efficiently support dynamic content, by using separate BVH data structures for individual geometries.*

The path trace integrator is unidirectional (rays issue from the eye) and uses quasi-Monte Carlo sampling, Russian-roulette termination, and local evaluation of direct illumination. Bidirectional path tracing, photon mapping, and multiple importance sampling are not supported. However, we believe this path tracer is sufficiently rich in features to be representative of a real-world renderer.

7.2 Vectorized Path Tracer

Single-ray SIMD traversal and intersection (Subsection 2.2) has the advantage that it can be used within a renderer which is otherwise scalar, and can achieve good utilization for a vector width of 4. However, single-ray methods are less effective for 8 and 16-wide vectors [Benthin et al. 2012]. Here, packet and hybrid packet / single-ray techniques are necessary to achieve high utilization, but require a renderer capable of generating multiple rays in parallel. Parallelism in the renderer is also needed to overcome Amdahl's Law. If shading and sampling together are roughly as expensive as ray traversal, even an infinitely fast traversal kernel cannot achieve more than a $2\times$ speedup in total per-frame performance.

Though a shading language compiler can be used to generate vectorized code that utilizes the Embree packet and hybrid kernels, we have implemented a vectorized path tracer using the Intel SPMD Program Compiler (ISPC) [Pharr and Mark 2012]. An approach based on the SPMD programming model in a high level language, avoids limiting the path tracer code to a given ISA or vector width. Further, ISPC is designed for use with CPUs and supports language and compiler features familiar to C++ developers such as recursion, function pointers, and linking multiple program objects. All shading and sampling code is implemented in ISPC, and the renderer accesses the Embree kernels via ISPC bindings in the Embree API.

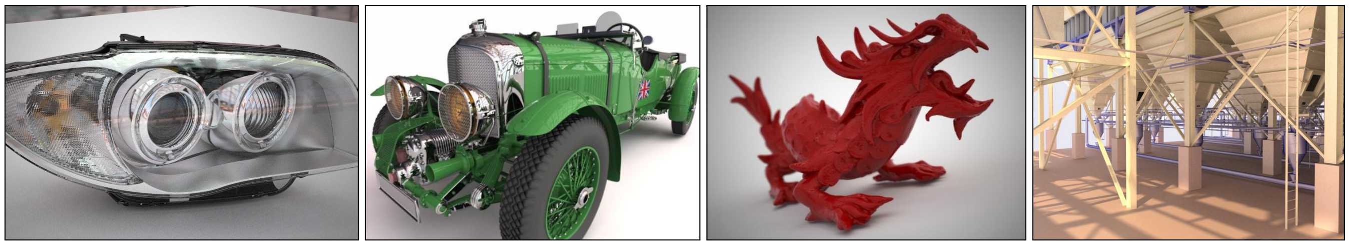

Figure 4: *Models used for evaluating the performance of Embree. From left to right: Headlight (800K triangles), Bentley (2.3M triangles), Dragon (7.4M triangles), Power Plant (12.7M triangles), and the Imperial Crown of Austria from Figure 1 (4.8M triangles). Embree is often used in production rendering environments with models of much greater complexity.*

The vectorized path tracer is used in our evaluation (Section 8) to compare the performance of the Embree packet / hybrid kernels with the single-ray SIMD kernels in the scalar path tracer. For this reason, we hold both path tracers to a similar design, code structure, and feature set. Since ISPC does not support C++, we emulate inheritance and virtual functions in the vectorized path tracer through the use of function pointers. In addition, we avoid ray reordering to increase SIMD utilization, but note that this technique is likely to offer a benefit similar to that in GPU-based ray tracing systems [Laine et al. 2013].

8 Evaluation

Evaluating the performance of rendering systems is challenging. The performance of a given renderer is highly sensitive to properties of the workload tested (e.g. mix of ray coherence). Further, Embree is not itself a complete system, and so the performance will vary based on properties of the application in which it is used (e.g. degree of vectorization). Finally, when comparing renderers on different hardware architectures, it can be difficult to isolate the performance benefits attributable to intrinsic advantages of the architecture from those due to the design of the renderer.

We address these challenges in three ways. We evaluate the benefit of specific design features in Embree (e.g. single-ray SIMD versus packet traversal) by comparing the relative performance of renderers built on Embree with and without these features enabled (Table 1, 2). We evaluate Embree relative to state-of-the-art GPU methods, by comparing the performance of a renderer built on Embree with a renderer built in OptiX, where the feature set of the two renderers is closely aligned (Table 3, 4). In all cases, we use workloads which are representative of the size, complexity, and illumination effects commonly used in professional rendering environments (Figure 4).

8.1 BVH Construction Performance

Embree BVH build times and build rates[3] are shown in Table 1 for the binned SAH kernel, and the Morton-code kernel. The reported times measure the interval between the point at which triangle data is available from the application, and the point at which a completed acceleration structure becomes available for ray traversal.

These results indicate that Embree can reach interactive build rates on CPUs for multi-million triangle scenes, and this performance compares favorably to existing methods on CPUs and GPUs. For example, Karras et al. [2013] report BVH build rates of between 30 and 40 million triangles per second using an NVIDIA GeForce GTX Titan GPU. These rates are similar to those for the Embree binned SAH build kernel on an Intel Xeon Phi coprocessor, despite the higher peak FLOPs available on the GPU.

The performance of BVH construction on the Haswell and Sandy-Bridge processors appears to be less competitive compared to that of the Intel Xeon Phi coprocessor. However, the results for the latter do not include the time to upload data across the PCI bus. Once this cost is included, the performance variance between the processors and coprocessor is reduced (Table 4).

8.2 Ray Traversal Performance

Table 2 reports traversal rates[3] for primary rays and full path traced illumination for a scalar renderer based on the Embree single-ray SIMD traversal kernels (Subsection 7.1), and a vectorized renderer based on the Embree hybrid traversal kernels (Subsection 7.2). For each experiment Embree selects the BVH subtype and triangle storage order for the given kernels, ray coherence, and scene geometry. These render times are only indicative of the results achievable with the Embree kernels. The actual performance in a given application will vary by renderer, architecture, and workload.

Vectorization in the renderer potentially improves the performance of shading and sampling code, and enables the use of the Embree packet and hybrid traversal kernels. The results in Table 2 illustrate both benefits, and indicate that a fully vectorized renderer is particularly important to exploiting the compute capability of architectures with wide vector widths, such as the Intel Xeon Phi coprocessor.

In some cases, design limitations or the level of effort required may make it infeasible to vectorize a renderer. Here, the Embree single-ray SIMD kernels may still be used to improve the performance of ray traversal. Compared to a scalar path tracer with scalar traversal, the single-ray SIMD kernels can yield a speedup in total frame time of 2 (on Haswell) to 8.6 (on the Intel Xeon Phi coprocessor).

8.3 Performance Relative to OptiX

State-of-the-art renderers on GPUs are known to achieve very high performance. For example, Karras et al. [2013] report 350 million rays per second for path tracing using an NVIDIA GeForce GTX Titan. However, the shading and sampling complexity included in these results, and the performance impact of less coherent ray distributions is unclear.

For this reason, we compare scalar and vectorized versions of a path tracer based on the Embree kernel framework, with a functionally-similar path tracer implemented in OptiX. All 3 renderers compute diffuse-only shading to minimize the impact of this application-dependent computation on render time. For this comparison we

[3] Embree results are reported where noted for the following systems: an Intel Core i7-4770 Haswell system (4 cores, 3.5GHz clock), a dual-socket Intel Xeon E5-2690 SandyBridge system (16 cores total, 2.9GHz clock), and an Intel Xeon Phi 7120 coprocessor (61 cores, 1.28GHz). Embree 2.2 is used, as compiled with Intel Composer XE 14.0.1 and ISPC 1.6.0.

[4] OptiX results are reported where noted for an NVIDIA GeForce GTX Titan with 6GB of memory, and OptiX version 3.5.1 built with CUDA 5.5.

	Embree Binned SAH Builder (Higher Quality)						Embree Morton-Code Builder (Higher Performance)					
Scene	Haswell		2× SandyBridge		Xeon Phi		Haswell		2× SandyBridge		Xeon Phi	
Headlight (0.8M)	142 ms	7M	82 ms	10M	23 ms	35M	24 ms	34M	11 ms	75M	6 ms	136M
Bentley (2.3M)	393 ms	6M	179 ms	13M	63 ms	37M	70 ms	33M	28 ms	83M	16 ms	147M
Crown (4.8M)	816 ms	6M	394 ms	12M	126 ms	39M	143 ms	33M	77 ms	63M	32 ms	150M
Dragon (7.4M)	1210 ms	6M	587 ms	13M	183 ms	40M	220 ms	33M	98 ms	75M	51 ms	141M
Power Plant (12.7M)	2501 ms	5M	1040 ms	12M	384 ms	33M	432 ms	30M	170 ms	75M	97 ms	131M
	(a)	(b)										

Table 1: *Embree BVH build times* (a) *and build rates in triangles per second* (b) *for the binned Surface Area Heuristic (SAH) kernel with spatial splits disabled (Subsection 5.2.1) and Morton-code kernel (Subsection 5.2.2). The former yields higher quality BVH structures for irregular, triangulated scenes, while the latter is much faster at the cost of lower quality hierarchies. Memory allocation time is excluded.*

	Embree Primary Rays (Including Simple Shading)									Embree Path Tracing (Including Full Shading)								
Scene	Haswell			2× SandyBridge			Xeon Phi			Haswell			2× SandyBridge			Xeon Phi		
Headlight	38M	89M	2.3×	110M	210M	1.9×	80M	374M	4.6×	12M	12M	1.0×	38M	34M	0.9×	31M	52M	1.7×
Bentley	52M	113M	2.2×	148M	284M	1.9×	99M	400M	4.0×	19M	23M	1.2×	61M	61M	1.0×	37M	96M	2.6×
Crown	59M	138M	2.3×	183M	349M	2.0×	105M	464M	4.4×	15M	17M	1.1×	46M	46M	1.0×	35M	84M	2.4×
Dragon	49M	96M	2.0×	131M	234M	1.8×	96M	339M	3.5×	20M	28M	1.4×	62M	75M	1.2×	47M	117M	2.3×
Power Plant	14M	27M	1.9×	33M	62M	1.9×	27M	93M	3.4×	9M	9M	1.0×	28M	29M	1.1×	22M	35M	1.6×
	(a)	(b)	(b / a)															

Table 2: *Render performance (rays per second) when using the Embree kernels in a complete path tracer. Performance is shown for primary rays only, and for full path tracing. For each test, results are reported for* (a) *a scalar C++ renderer with the Embree single-ray SIMD kernels (Subsection 7.1) and* (b) *a vectorized renderer written in ISPC with the Embree hybrid packet / single-ray SIMD kernel (Subsection 7.2). Both use a BVH produced by the binned SAH build kernel with spatial splits disabled (Table 1). Performance is measured as the total rays traced divided by total frame time, including sampling (16 samples per pixel) and shading (typically 30 to 50% of the total frame time). All images were rendered at* 1920×1080 *pixel resolution.*

	Embree Single-Ray		Embree Hybrid		OptiX	
Scene	2× SandyBridge		Xeon Phi		GTX Titan	
Headlight	66M	70M	134M	136M	72M	80M
Bentley	54M	61M	90M	98M	53M	56M
Crown	45M	48M	64M	69M	37M	36M
Dragon	47M	50M	84M	85M	45M	50M
Power Plant	32M	45M	43M	59M	44M	41M
	(a)	(b)			(c)	(d)

Table 3: *Performance in rays per second for a scalar renderer with the Embree single-ray SIMD kernels (Subsections 5.1.1, 5.1.2), a parallel renderer written in ISPC with the Embree hybrid packet / single-ray SIMD kernel (Subsection 5.1.4), and an OptiX renderer. All three renderers are diffuse-only path tracers. Embree results are shown for a BVH without* (a) *and with* (b) *spatial splits. Similarly, OptiX results are shown for a TRBVH* (c) *and a SBVH* (d).

	Embree Binned SAH Builder				OptiX	
Scene	2× SandyBridge		Xeon Phi		GTX Titan	
Headlight	0.1s	0.2s	0.3s	0.3s	0.4s	6.5s
Bentley	0.3s	0.4s	1.5s	1.6s	0.7s	14.4s
Crown	0.5s	0.9s	1.6s	1.7s	1.1s	36.5s
Dragon	0.7s	1.4s	1.5s	1.6s	1.4s	44.7s
Power Plant	1.3s	2.8s	2.9s	3.2s	2.5s	113.3s
	(a)	(b)			(c)	(d)

Table 4: *Start times for the Embree and OptiX based renderers, including API calls, memory allocation, data upload (for the Intel Xeon Phi coprocessor and NVIDIA GeForce GTX Titan), and BVH construction. Embree times are reported for BVH structures built without* (a) *and with* (b) *spatial splits. OptiX times are reported for a GPU build of a BVH without spatial splits* (c) *and a single-threaded CPU build of a BVH with spatial splits* (d).

have enabled the fast and high quality TRBVH in Optix and spatial splits in the Embree BVH build kernel. The performance in rays-per-second for the 3 renderers is shown in Table 3, including shading and sampling[3,4]. Startup times including BVH construction and data upload (but not scene file parsing) are shown in Table 4. In all cases the Embree and OptiX renderers perform comparably.

9 Conclusion

In this paper, we describe Embree, a kernel framework for efficient ray tracing on x86 CPUs. Embree is directed at professional rendering environments in which scenes with high geometric complexity and indirect illumination are the norm rather than the exception. To address this focus, Embree provides a set of commonly used kernels optimized for different ISA vector widths, workloads (e.g. coherent and incoherent ray distributions, static and dynamic scenes), and application-specific priorities (e.g. maximal performance or minimal memory usage). Renderers built on these kernels can achieve BVH build and ray traversal performance comparable to (and often higher than) existing methods on any current CPU or GPU.

Further, this kernel-level approach is broadly applicable and avoids placing limits on the design of a complete rendering application. For example, the Embree kernels enable maximal performance in renderers capable of tracing multiple rays in parallel, but there is no requirement that the renderer be parallelized. The Embree single-ray SIMD kernels can be used in a scalar renderer, via the same API. By seamlessly supporting both options, Embree enables developers to incrementally move from scalar to fully parallel rendering.

However, this flexibility comes with a tradeoff. Embree is not itself a stand-alone global illumination system, nor a drop-in replacement for existing rendering engines. As a consequence, Embree cannot address system-level optimizations such as ray reordering, adaptive sampling and reconstruction, or paging of large model data between a host and an accelerator (unlike OpenRT or OptiX). In addition, Embree lacks built-in support for texture mapping and volume rendering. For these reasons, we do not claim Embree is the best choice for every CPU based renderer. Rather, Embree offers a combination of performance, flexibility, and ease of use that is potentially useful in many applications.

Acknowledgements

The bedroom scene from the movie "Peabody & Sherman" is courtesy of DreamWorks Animation SKG. The model of the Imperial Crown of Austria is provided by Martin Lubich[5]. The White Room model is courtesy of Jay Hardy.

We would like to thank Ram Ramanujam, Paresh Pattani, Jim Hurley, Bill Mark, Chris Seitz, and Joe Curley for their guidance and backing. We thank the Intel ISPC compiler team for their continuing support. We also thank the many ISVs and users of Embree (particularly Louis Feng, Evan Smyth, and Eric Tabellion) for their valuable feeback and contributions.

References

AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics*, 145–149.

BENTHIN, C., AND WALD, I. 2009. Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing. In *Proceedings of High-Performance Graphics*, 135–144.

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

BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 187–196.

DAMMERTZ, H., HANIKA, J., AND KELLER, A. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. In *Proceedings of the 19th Eurographics Conference on Rendering*, 1225–1234.

ERNST, M., AND GREINER, G. 2008. Multi Bounding Volume Hierarchies. In *Proceedings of the IEEE / Eurographics Symposium on Interactive Ray Tracing*, 35–40.

GLASSNER, A. 1989. *An Introduction to Ray Tracing*. Morgan Kaufmann.

GRÜNSCHLOSS, L., STICH, M., NAWAZ, S., AND KELLER, A. 2011. MSBVH: An Efficient Acceleration Data Structure for Ray Traced Motion Blur. In *Proceedings of High-Performance Graphics*, 65–70.

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

KARRAS, T., AND AILA, T. 2013. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics*, 89–99.

KENSLER, A., AND SHIRLEY, P. 2006. Optimizing Ray-Triangle Intersection via Automated Search. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 33–38.

KENSLER, A. 2008. Tree Rotations for Improving Bounding Volume Hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 73–76.

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of High-Performance Graphics*, 137–143.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. In *Computer Graphics Forum: Proceedings of Eurographics*, 375–384.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers*, 66:1–66:13.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufman.

PHARR, M., AND MARK, B. 2012. ISPC - A SPMD compiler for high-performance CPU programming. In *Proceedings of Innovative Parallel Computing*, 1–13.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-Level Ray Tracing Algorithm. In *ACM SIGGRAPH 2005 Papers*, 1176–1185.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics*, 7–13.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2002. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Tech. rep., Saarland University. Available at http://graphics.cs.uni-sb.de/Publications.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 11–20.

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 IEEE / Eurographics Symposium on Interactive Ray Tracing*, 49–57.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University.

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

WALD, I. 2012. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1, 47–57.

[5] `http://www.loramel.net`