Embree: Photo-Realistic Ray Tracing Kernels

User's Guide

Software Version 1.0 beta, June 2011

Abstract

Embree is a collection of high-performance ray tracing kernels, developed at Intel Labs. The kernels are optimized for photo-realistic rendering on the latest Intel® processors with support for SSE and AVX instructions. In addition to the ray tracing kernels, Embree provides an example photo-realistic rendering engine. The renderer is not intended to be a complete solution. Instead, it serves two purposes: (a) demonstrating how the ray tracing kernels are used in practice, and (b) measuring the performance of the kernels in a realistic application scenario.

Embree is designed for Monte Carlo ray tracing algorithms, where the vast majority of rays are incoherent. The specific single-ray traversal kernels in Embree provide the best performance in this scenario and they are very easy to integrate into existing applications. The kernels can be used to develop new rendering engines on top of them, to replace the core of an existing renderer or simply as a benchmark. Embree is released as open source under the Apache 2.0 license.

This document gives an overview of how the Embree code base is structured and how the example renderer is compiled and run.

1 Introduction

Monte Carlo ray tracing is the best know method for photo-realistic rendering. The algorithm is popular for professional applications, because it is robust, it supports physically accurate rendering, and it performs very well on modern multi-core processors. Typical applications include virtual prototyping, architectural visualization, and movie production.

A major part of the runtime of a Monte Carlo ray tracer is consumed by geometric ray queries, i.e. finding the closest intersection along a ray, or determining if a ray hits any object at all. Embree provides the technology required to (a) efficiently perform these queries and to (b) efficiently build the underlying acceleration data structures. Embree may be used as the foundation of a new renderer or to improve or replace the ray tracing kernels of an existing application.

Embree includes an example path tracing renderer implementing an API and a few simple test scenes. This makes it possible to do performance evaluations and comparisons in three different ways:

- Plugging Embree's ray tracing kernels into an existing application
- Plugging the ray tracing kernels of an existing application into Embree
- Attaching an application to the Embree Rendering API

2 Getting Started

Embree runs on Windows, Linux and MacOSX, each in 32bit and 64bit modes. The code compiles with the Intel Compiler, the Microsoft Compiler and with GCC. Using the Intel Compiler improves performance by approximately 10%. Performance also varies across different operating systems. Embree is optimized for Intel CPUs supporting SSSE3, SSE4.1, SSE4.2 and AVX.

2.1 Compiling Embree

For compilation under Linux and MacOSX you have to install CMake (for compilation) the developer version of GLUT (for display) and we recommend installing the ImageMagick and OpenEXR developer packages (for reading and writing images). To compile the code using CMake create a build directory such as `embree/build` and execute `ccmake ..` inside this directory. This will open a configuration dialog where you should set the build mode to "Release", the SSE version to either SSSE3, SSE4.1, SSE4.2, or AVX, and possibly enable the ICC compiler for better performance. Press `c` (for configure) and `g` (for generate) to generate a Makefile and leave the configuration. The code can now be compiled by executing `make`. The executable `embree` will be generated in the build folder.

For compilation under Windows we recommend using the Visual Studio 2008 or Visual Studio 2010 solution files. You can switch between the Microsoft Compiler and the Intel Compiler by right clicking on the project and selecting the compiler. The project compiles with both compilers in 32bit and 64bit mode. We recommend using 64bit mode and the Intel Compiler for best performance. When using the Microsoft Compiler, SSE4 is enabled by default in the code base. Disabling this default setting by removing the `__SSE4_2__` define in `common/sys/platform.h` is necessary when SSE4 is not supported on your platform. Before you can run the application you have to install the GLUT library provided in the `app/freeglut` folder. Copy the 64bit DLL into the `x64/Release` and `x64/Debug` folder, and the 32bit DLL into the `Win32/Release` and `Win32/Debug` folder.

2.2 Running Embree

This section describes Embree's most important command line parameters. Execute `embree -help` for a complete list of parameters. Embree ships with a few simple test scenes, each consisting of a scene file (.xml or .obj) and an Embree command script file (.ecs). The command script file contains command line parameters that set the camera parameters, lights and render settings.

The following command line will render the Cornell box scene with 16 samples per pixel and write the resulting image to the file `cb.tga` in the current directory:

```
embree -c models/cornell_box.ecs -spp 16 -o cb.tga
```

You might have to adjust the path to the executable or model file for your system. To interactively display the same scene, enter the following command:

```
embree -c models/cornell_box.ecs
```

A window will open and you can control the camera using the mouse and keyboard. Pressing `c` in interactive mode outputs the current camera parameters, pressing `r` enables or disables the progressive refinement mode.

2.3 Camera Navigation

The navigation in the interactive display mode follows the camera orbit model, where the camera revolves around the current center of interest. The camera navigation assumes the y-axis to point upwards. If your scene is modeled using the z-axis as up axis we recommend rotating the scene.

LMB	Rotate around center of interest
MMB	Pan
RMB	Dolly (move camera closer or away from center of interest)
Strg+LMB	Pick center of interest
Strg+Shift+LMB	Pick focal distance
Alt+LMB	Roll camera around view direction
L	Decrease lens radius by one world space unit
Shift+L	Increase lens radius by one world space unit

3 System Overview

This section gives an overview of the structure of the code. Embree consists of four separate parts.

A *base library* in the `embree/common` folder. It contains general purpose functionality, such as a short vector class, SIMD wrapper classes, and some platform abstractions. All other parts of the system use this base library.

The *ray tracing kernels* are isolated from the rest of the system in the `embree/rtcore` folder. It implements different acceleration structure builders and traverser. The kernels are standalone with dependencies only to the base library. For more information on the kernels see Section 4.

The *rendering system* is contained in the `embree/renderer` directory. It is structured into different components and based on the concepts of the PBRT renderer[1]. The renderer provides an

API layer located in `embree/renderer/api/api.h`. The frontend application implements a useful C++ wrapper for this API. For more information on the rendering system see Section 5.

The *frontend application* is located in the `embree/app` folder. It is responsible for parsing scenes and displaying the image. The frontend is less important for the intended usage scenarios of Embree, thus it is not described in detail.

4 Ray Tracing Kernels

The ray tracing kernels are the main contribution of Embree and they provide efficient implementations of two different spatial index structures:

BVH2: Axis aligned BVH with a branching factor of two.

BVH4: Axis aligned BVH with a branching factor of four.

The builders and traversers of all data structures are carefully optimized for the SSE (and AVX) instruction set.

4.1 BVH4

This data structure implements a BVH with a branching factor of four. Each node stores the bounding box and pointers to its four children. Chunks of four triangles are stored and intersected at the leaf nodes with SSE. Nodes as well as leaves are typically not completely filled, resulting in a 20% memory overhead. In average we see a memory consumption of 76 bytes per triangle for the entire data structure.

4.1.1 Object Split Builder

The object split builder is a highly optimized and multi-threaded builder producing no primitive duplications. On a quad-core 2nd generation Intel® Core™ system a build performance of about 8-9 million triangles per second is achievable.

The builder is a top-down builder using an SAH binning approach to find the best object partitioning. At every node, we split the child that gives the highest SAH gain until the node is completely filled and then descend into its children.

The builder implements 3 different strategies for different granularities:

- Small size jobs are completed in a single thread to avoid threading overhead.
- Medium size jobs are split by a single thread into 4 sub-jobs.
- Large size jobs are split in a multi-threaded way into 4 sub-jobs.

The first two strategies are important to gain good performance. The 3rd strategy can increase performance by additional 10% when building large scenes. We recommend using this builder for very large scenes or for previews where build time can become a bottleneck.

4.1.2 Spatial Split Builder

The spatial split builder is also multi-threaded but produces multiply referenced primitives. In addition to object splits, the spatial split builder tests spatial splits in the center of each dimension, and selects the better of these two options. The maximum number of primitive replications can be accurately controlled by changing the replication factor. In order to distribute the replication evenly over the scene, the builder operated in breadth-first order.

Compared to the object split builder, build performance is about 5x slower, but render performance can be 2x better for problematic scenes. In particular, scenes with long diagonal triangles benefit from this build procedure. The implementation of this builder does not natively build a BVH4, but a BVH2 which is then converted. Nevertheless, we recommend using this builder for scenes that contain many non-axis aligned triangles (some architectural scenes fall in this category).

4.1.3 Traversal

The traversal routine is hand tuned code implementing a standard recursive traversal of the tree. We intersect the single ray with the four boxes in SSE and descend into the hit children in front to back order. At the leaf nodes the ray is intersected with chunks of 4 triangles using SSE code. The next sections describes some of the optimizations we implemented to achieve higher performance.

Load Near/Far Plane: For efficient ray/box intersection, we implement an optimization to switch loading of the near and far plane in each dimension based on the direction sign of the ray. This reduces the data dependency latency of the kernel compared to the alternative approach of selecting the near/far plane by additional min/max operations.

0 Children Hit: We optimize the 0-children-hit case, by doing an early pop optimization. We load the next node from the stack into registers in parallel to the ray/box intersection, which makes a pop essentially free (apart from branch mispredictions).

1 Child Hit: We optimize the likely 1-child-hit case by avoiding a push/pop sequence and keeping the ID of the next node to traverse in registers.

2 Children Hit: We also optimize the 2-children-hit case, by pushing the farther child and continuing with the closer child. Optimizing the 0, 1 and 2 children hit cases is important as they occur with 90% probability.

Cull at Pop: We store the far-distance of nodes on the stack. This allows us to cull a popped node early, when a closer intersection was found after its push event.

Sorted Push: When more than two children are hit we first push the nodes onto the stack, and then sort the stack entries before we pop the closest one. To push the nodes, we use `bsf` and `btc` instructions to iterate through the hit bit-vector until no more hit child is found. This iteration is implemented as an unrolled loop.

4.2 BVH2

The BVH2 is a binary BVH with a special node layout to make the traversal SSE friendly. For a node, each dimension stores the lower and upper bounds of the left and right child in a single SSE vector. This allows for an efficient swap of the lower and upper bound for rays going from "right to left" and processing of the ray/plane intersections in SSE. Chunks of four triangles are stored and intersected at the leaf nodes with SSE code.

The BVH2 traverser is as fast as the BVH4 if hyper-threading is disabled. For enabled hyper-threading, the BVH4 outperforms the binary BVH.

4.3 Extracting the Ray Tracing Kernels

The most important usage scenario of Embree is extracting its ray tracing kernels and plugging them into an existing codebase, either for performance comparisons or to replace the ray tracing kernels of the existing application with Embree's kernels. The dependencies of the kernels and their interfaces are explained in this section.

Embree's ray tracing kernels, located in the `embree/rtcore` folder, are well encapsulated. They only have dependencies to the `embree/common/sys` (system abstraction), `embree/common/simd` (SIMD wrappers), and `embree/common/math` (short vector library) folders. These folders can optionally be copied into the `rtcore` folder to obtain a self contained ray tracing kernel. For compilation of the kernels the include-path has to be set to the `rtcore` folder. We recommend compiling the kernels into a dynamic link library.

The ray tracing kernels use a tasking system implemented in `sys/tasking.cpp` solely for parallel building of data structures. Before using the ray tracing kernels the tasking system must be initialized by calling the `TaskScheduler::init` function and cleaned up at application exit using the `.TaskScheduler::cleanup` function. Using the tasking system in existing code will generate one additional thread per system hyper-thread. These threads are very active as long as tasks need to get processed but sleeping when not required. Thus they should not cause any performance problems in the existing codebase. If desired the tasking system can be modified in such a way that the `TaskScheduler::run` function is called from threads available in the existing application when a builder is active.

The programming interface to the ray tracing kernels is located in `rtcore/rtcore.h`, and the ray and hit data structures are defined in `rtcore/ray.h` and `rtcore/hit.h`. There are two different interfaces, one for building data structures and one for tracing rays.

Data structures are built by calling the `rtcCreateAccel` function with a list of build triangles. A build triangle is a special self contained triangle data structure defined in `rtcore.h`.

The intersector interface returned by the `rtcCreateAccel` function is used to trace rays using the `intersect` or `occluded` function. The `intersect` function traverses a ray through the spatial index structure and returns the closest hit found. The `occluded` function tests if the ray is occluded by any geometry along the ray. A ray in the system is actually a ray segment defined by position, direction, and a parameter interval.

One issue when using the ray tracing kernels in an existing code base might be that different short vector and SIMD wrapper libraries are used in each code base. We recommend *not* modifying the Embree core to use any other short vector or SIMD wrapper library. Instead, a wrapper layer should be written around the `rtcore` interface to obtain an interface that uses the existing short vector library. If possible pointer casts should be used in this layer instead of copying actual data.

This is of particular importance for the trace functions where no overhead should be introduced at all. For instance, if the memory layout of the ray and hit structures can be matched between Embree and the existing code base, a simple pointer cast in the wrapper layer is sufficient.

5 Renderer

The purpose of the Embree renderer is to provide an environment for evaluating the traversal kernels in a flexible framework. It also serves as an example of how the ray tracing kernels are used in a rendering system. It is important to note that the renderer is *not* intended as a complete solution. Its feature set is limited. Therefore it is probably not suitable for replacing an existing rendering engine. It does, however, serve as an example of how an efficient rendering engine is designed and implemented.

Our design is inspired by the PBRT renderer[1] which implements a clean separation of different aspects of the system into modular components. Like PBRT, Embree supports programmable lights, BRDFs, materials, integrators, renderers, and samplers. All these different components communicate through well defined interfaces. This allows for modifying and extending them separately from other modules of the system, for example materials and lights can be added independently of the renderers.

The path tracer together with a variety of implemented surface reflectance models supports the generation of a wide range of non-trivial ray distributions. These range from relatively coherent reflections and transmissions from *metal* and *thin glass* BRDFs, to highly incoherent diffuse bounces and soft shadows from HDR environment lighting. Deep specular and glossy reflections and refractions in glass bodies, such as headlights, are supported as well.

The renderer is partitioned into several components. Each component is located in its own folder, see Table below. Each of these folders contains an interface file named similar to the folder itself, e.g. the file `brdfs/brdf.h` defines the interface to the BRDFs, and `integrators/integrator.h` the interface to the integrators.

API	renderer/api
BRDFs	renderer/brdfs
Cameras	renderer/cameras
Pixel Filters	renderer/filters
Integrators	renderer/integrators
Lights	renderer/lights
Materials	renderer/materials
Renderers	renderer/renderers
Samplers	renderer/samplers
Shapes	renderer/shapes
Textures	renderer/textures

The Embree renderer provides an API for ease of use. It needs to be initialized by calling `rtInit()` at the beginning and `rtExit()` at the end of your application. The API is completely functional, meaning that objects *cannot* be modified. Handles are references to objects, and objects are internally reference counted, thus destroyed when no longer needed. Calling `rtDelete` merely deletes the handle, not the object it references. Calling one of the `rtNew*` functions creates a new handle referencing a new object. The `rtSet*` functions buffer parameters to be set inside the handle. A subsequent call to `rtCommit` will set the handle reference to a new object with these new parameters. The original object is not changed by this process. To find the parameters supported by a specific object see the header file of the object implementation. The semantics of modifying an object A used by another object B can only be achieved by creating A' and a new B' that uses A'.

The renderer splits the frame buffer into a set of tiles and processes them with multiple threads. All threads are completely independent. A renderer typically uses an integrator to integrate the incident radiance at a given point and direction.

The current implementation provides a path tracing integrator. It is very efficient when the scene is lit with a low-frequency HDR environment map. However, it is *not* designed for efficient handling of scenes with many light sources or strong caustics. A more sophisticated integrator is required for such scenarios.

As mentioned earlier, the primary intent of Embree is to provide highly optimized ray tracing kernels, not a complete rendering solution. To get a deeper understanding of the design and implementation of complete rendering systems, we recommend reading that book PBRT[1]. Embree adopts many of the fundamental concepts in this book.

6 Bibliography

[1]Matt Pharr and Greg Humphreys: *Physically Based Rendering: From Theory To Implementation.* Morgan Kaufmann, 2nd revised edition, 2010.