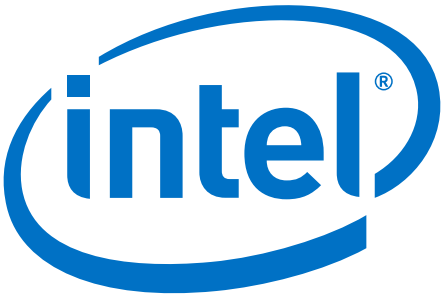

EMBREE
HIGH PERFORMANCE RAY TRACING KERNELS

Version 2.16.0
May 17, 2017

Contents

Chapter 1
Embree Overview

Embree is a collection of high-performance ray tracing kernels, developed at Intel. The target user of Embree are graphics application engineers that want to improve the performance of their application by leveraging the optimized ray tracing kernels of Embree. The kernels are optimized for photo-realistic rendering on the latest Intel® processors with support for SSE, AVX, AVX2, and AVX-512. Embree supports runtime code selection to choose the traversal and build algorithms that best matches the instruction set of your CPU. We recommend using Embree through its API to get the highest benefit from future improvements. Embree is released as Open Source under the Apache 2.0 license.

Embree supports applications written with the Intel SPMD Program Compiler (ISPC, `https://ispc.github.io/`) by also providing an ISPC interface to the core ray tracing algorithms. This makes it possible to write a renderer in ISPC that leverages SSE, AVX, AVX2, and AVX-512 without any code change. ISPC also supports runtime code selection, thus ISPC will select the best code path for your application, while Embree selects the optimal code path for the ray tracing algorithms.

Embree contains algorithms optimized for incoherent workloads (e.g. Monte Carlo ray tracing algorithms) and coherent workloads (e.g. primary visibility and hard shadow rays). For standard CPUs, the single-ray traversal kernels in Embree provide the best performance for incoherent workloads and are very easy to integrate into existing rendering applications. For AVX-512 enabled machines, a renderer written in ISPC using the default hybrid ray/packet traversal algorithms have shown to perform best, but requires writing the renderer in ISPC. In general for coherent workloads, ISPC outperforms the single ray mode on each platform. Embree also supports dynamic scenes by implementing high performance two-level spatial index structure construction algorithms.

In addition to the ray tracing kernels, Embree provides some tutorials to demonstrate how to use the Embree API. The example photorealistic renderer that was originally included in the Embree kernel package is now available in a separate GIT repository (see Embree Example Renderer).

Supported Platforms

Embree supports Windows (32 bit and 64 bit), Linux (64 bit) and Mac OS X (64 bit). The code compiles with the Intel Compiler, GCC, Clang and the Microsoft Compiler.

Using the Intel Compiler improves performance by approximately 10%. Performance also varies across different operating systems, with Linux typically performing best as it supports transparently transitioning to 2MB pages.

Embree is optimized for Intel CPUs supporting SSE, AVX, AVX2, and AVX-512 instructions, and requires at least a CPU with support for SSE2.

Embree Support and Contact

If you encounter bugs please report them via Embree's GitHub Issue Tracker.

For questions please write us at `embree_support@intel.com`.

To receive notifications of updates and new features of Embree please subscribe to the Embree mailing list.

Version History

New Features in Embree 2.16.0

- Improved multi-segment motion blur support for scenes with different number of time steps per mesh.
- New top level BVH builder that improves build times and BVH quality of two-level BVHs.
- Added support to enable only a single ISA. Previously code was always compiled for SSE2.
- Improved single ray tracing performance for incoherent rays on AVX512 architectures by 5-10%.
- Improved packet/hybrid ray tracing performance for incoherent rays on AVX512 architectures by 10-30%.
- Improved stream ray tracing performance for coherent rays in structure-of-pointers layout by 40-70%.
- BVH builder for compact scenes of triangles and quads needs essentially no temporary memory anymore. This doubles the maximal scene size that can be rendered in compact mode.
- Triangles no longer store the geometry normal in fast/default mode which reduces memory consumption by up to 20%.
- Compact mode uses BVH4 now consistently which reduces memory consumption by up to 10%.
- Reduced memory consumption for small scenes (of 10k-100k primitives) and dynamic scenes.
- Improved performance of user geometries and instances through BVH8 support.
- The API supports now specifying the geometry ID of a geometry at construction time. This way matching the geometry ID used by Embree and the application is simplified.
- Fixed a bug that would have caused a failure of the BVH builder for dynamic scenes when run on a machine with more then 1000 threads.
- Fixed a bug that could have been triggered when reaching the maximal number of mappings under Linux (`vm.max_map_count`). This could have happened when creating a large number of small static scenes.
- Added huge page support for Windows and MacOSX (experimental).
- Added support for Visual Studio 2017.
- Removed support for Visual Studio 2012.
- Precompiled binaries now require a CPU supporting at least the SSE4.2 ISA.
- We no longer provide precompiled binaries for 32 bit on Windows.
- Under Windows one now has to use the platform toolset option in CMake to switch to Clang or the Intel® Compiler.
- Fixed a bug for subdivision meshes when using the incoherent scene flag.
- Fixed a bug in the line geometry intersection, that caused reporting an invalid line segment intersection with primID -1.
- Buffer stride for vertex buffers of different time steps of triangle and quad meshes have to be identical now.

- Fixed a bug in the curve geometry intersection code when passed a perfect cylinder.

New Features in Embree 2.15.0

- Added `rtcCommitJoin` mode that allows thread to join a build operation. When using the internal tasking system this allows Embree to solely use the threads that called `rtcCommitJoin` to build the scene, while previously also normal worker threads participated in the build. You should no longer use `rtcCommit` to join a build.
- Added `rtcDeviceSetErrorFunction2` API call, which sets an error callback function which additionally gets passed a user provided pointer (`rtcDeviceSetErrorFunction` is now deprecated).
- Added `rtcDeviceSetMemoryMonitorFunction2` API call, which sets a memory monitor callback function which additionally get passed a user provided pointer. (`rtcDeviceSetMemoryMonitorFunction` is now deprecated).
- Build performance for hair geometry improved by up to 2×.
- Standard BVH build performance increased by 5%.
- Added API extension to use internal Morton-code based builder, the standard binned-SAH builder, and the spatial split-based SAH builder.
- Added support for BSpline hair and curves. Embree uses either the Bézier or BSpline basis internally, and converts other curves, which requires more memory during rendering. For reduced memory consumption set the `EMBREE_NATIVE_SPLINE_BASIS` to the basis your application uses (which is set to `BEZIER` by default).
- Setting the number of threads through `tbb::taskscheduler_init` object on the application side is now working properly.
- Windows and Linux releases are build using AVX-512 support.
- Implemented hybrid traversal for hair and line segments for improved ray packet performance.
- AVX-512 code compiles with Clang 4.0.0
- Fixed crash when ray packets were disabled in CMake.

New Features in Embree 2.14.0

- Added `ignore_config_files` option to init flags that allows the application to ignore Embree configuration files.
- Face-varying interpolation is now supported for subdivision surfaces.
- Up to 16 user vertex buffers are supported for vertex attribute interpolation.
- Deprecated `rtcSetBoundaryMode` function, please use the new `rtcSetSubdivisionMode` function.
- Added `RTC_SUBDIV_PIN_BOUNDARY` mode for handling boundaries of subdivision meshes.
- Added `RTC_SUBDIV_PIN_ALL` mode to enforce linear interpolation for subdivision meshes.
- Optimized object generation performance for dynamic scenes.
- Reduced memory consumption when using lots of small dynamic objects.
- Fixed bug for subdivision surfaces using low tessellation rates.
- Hair geometry now uses a new ribbon intersector that intersects with ray-facing quads. The new intersector also returns the v-coordinate of the hair intersection, and fixes artefacts at junction points between segments, at the cost of a small performance hit.
- Added `rtcSetBuffer2` function, that additionally gets the number of elements of a buffer. In dynamic scenes, this function allows to quickly

change buffer sizes, making it possible to change the number of primitives of a mesh or the number of crease features for subdivision surfaces.
- Added simple 'viewer_anim' tutorial for rendering key frame animations and 'buildbench' for measuring BVH (re-)build performance for static and dynamic scenes.
- Added more AVX-512 optimizations for future architectures.

New Features in Embree 2.13.0

- Improved performance for compact (but not robust) scenes.
- Added robust mode for motion blurred triangles and quads.
- Added fast dynamic mode for user geometries.
- Up to 20% faster BVH build performance on the second generation Intel® Xeon Phi™ processor codenamed Knights Landing.
- Improved quality of the spatial split builder.
- Improved performance for coherent streams of ray packets (SOA layout), e.g. for fast primary visibility.
- Various bug fixes in tessellation cache, quad-based spatial split builder, etc.

New Features in Embree 2.12.0

- Added support for multi-segment motion blur for all primitive types.
- API support for stream of pointers to single rays (`rtcIntersect1Mp` and `rtcOccluded1Mp`)
- Improved BVH refitting performance for dynamic scenes.
- Improved high-quality mode for quads (added spatial split builder for quads)
- Faster dynamic scenes for triangle and quad-based meshes on AVX2 enabled machines.
- Performance and correctness bugfix in optimization for streams of coherent (single) rays.
- Fixed large memory consumption (issue introduced in Embree v2.11.0). If you use Embree v2.11.0 please upgrade to Embree v2.12.0.
- Reduced memory consumption for dynamic scenes containing small meshes.
- Added support to start and affinitize TBB worker threads by passing "`start_threads=1,set_affinity=1`" to `rtcNewDevice`. These settings are recommended on systems with a high thread count.
- `rtcInterpolate2` can now be called within a displacement shader.
- Added initial support for Microsoft's Parallel Pattern Library (PPL) as tasking system alternative (for optimal performance TBB is highly recommended).
- Updated to TBB 2017 which is released under the Apache v2.0 license.
- Dropped support for Visual Studio 2012 Win32 compiler. Visual Studio 2012 x64 is still supported.

New Features in Embree 2.11.0

- Improved performance for streams of coherent (single) rays flagged with `RTC_INTERSECT_COHERENT`. For such coherent ray streams, e.g. primary rays, the performance typically improves by 1.3–2×.
- New spatial split BVH builder for triangles, which is 2–6× faster than the previous version and more memory conservative.
- Improved performance and scalability of all standard BVH builders on systems with large core counts.
- Fixed `rtcGetBounds` for motion blur scenes.
- Thread affinity is now on by default when running on the latest Intel® Xeon Phi™ processor.

- Added AVX-512 support for future Intel® Xeon processors.

New Features in Embree 2.10.0

- Added a new curve geometry which renders the sweep surface of a circle along a Bézier curve.
- Intersection filters can update the `tfar` ray distance.
- Geometry types can get disabled at compile time.
- Modified and extended the ray stream API.
- Added new callback mechanism for the ray stream API.
- Improved ray stream performance (up to 5–10%).
- Up to 20% faster morton builder on machines with large core counts.
- Lots of optimizations for the second generation Intel® Xeon Phi™ processor codenamed Knights Landing.
- Added experimental support for compressed BVH nodes (reduces node size to 56–62% of uncompressed size). Compression introduces a typical performance overhead of ~10%.
- Bugfix in backface culling mode. We do now properly cull the backfaces and not the frontfaces.
- Feature freeze for the first generation Intel® Xeon Phi™ coprocessor codenamed Knights Corner. We will still maintain and add bug fixes to Embree v2.9.0, but Embree 2.10 and future versions will no longer support it.

New Features in Embree 2.9.0

- Improved shadow ray performance (10–100% depending on the scene).
- Added initial support for ray streams (10–30% higher performance depending on ray coherence in the stream).
- Added support to calculate second order derivatives using the `rtcInterpolate2` function.
- Changed the parametrization for triangular subdivision faces to the same scheme used for pentagons.
- Added support to query the Embree configuration using the `rtcDeviceGetParameter` function.

New Features in Embree 2.8.1

- Added support for setting per geometry tessellation rate (supported for subdivision and Bézier geometries).
- Added support for motion blurred instances.

New Features in Embree 2.8.0

- Added support for line segment geometry.
- Added support for quad geometry (replaces triangle-pairs feature).
- Added support for linear motion blur of user geometries.
- Improved performance through AVX-512 optimizations.
- Improved performance of lazy scene build (when using TBB 4.4 update 2).
- Improved performance through huge page support under linux.

New Features in Embree 2.7.1

- Internal tasking system supports cancellation of build operations.
- ISPC mode for robust and compact scenes got significantly faster (implemented hybrid traversal for bvh4.triangle4v and bvh4.triangle4i).
- Hair rendering got faster as we fixed some issues with the SAH heuristic cost factors.

- BVH8 got slight faster for single ray traversal (improved sorting when hitting more than 4 boxes).
- BVH build performance got up to 30% faster on CPUs with high core counts (improved parallel partition code).
- High quality build mode again working properly (spatial splits had been deactivated in v2.7.0 due to some bug).
- Support for merging two adjacent triangles sharing a common edge into a triangle-pair primitive (can reduce memory consumption and BVH build times by up to 50% for mostly quad-based input meshes).
- Internal cleanups (reduced number of traversal kernels by more templating)
- Reduced stack size requirements of BVH builders.
- Fixed crash for dynamic scenes, triggered by deleting all geometries from the scene.

New Features in Embree 2.7.0

- Added device concept to Embree to allow different components of an application to use Embree without interfering with each other.
- Fixed memory leak in twolevel builder used for dynamic scenes.
- Fixed bug in tesselation cache that caused crashes for subdivision surfaces.
- Fixed bug in internal task scheduler that caused deadlocks when using `rtcCommitThread`.
- Improved hit-distance accuracy for thin triangles in robust mode.
- Added support to disable ray packet support in cmake.

New Features in Embree 2.6.2

- Fixed bug triggered by instantiating motion blur geometry.
- Fixed bug in hit UV coordinates of static subdivision geometries.
- Performance improvements when only changing tessellation levels for subdivision geometry per frame.
- Added ray packet intersectors for subdivision geometry, resulting in improved performance for coherent rays.
- Reduced virtual address space usage for static geometries.
- Fixed some AVX2 code paths when compiling with GCC or Clang.
- Bugfix for subdiv patches with non-matching winding order.
- Bugfix in ISA detection of AVX-512.

New Features in Embree 2.6.1

- Major performance improvements for ray tracing subdivision surfaces, e.g. up to 2× faster for scenes where only the tessellation levels are changing per frame, and up to 3× faster for scenes with lots of crease features
- Initial support for architectures supporting the new 16-wide AVX-512 ISA
- Implemented intersection filter callback support for subdivision surfaces
- Added `RTC_IGNORE_INVALID_RAYS` CMake option which makes the ray intersectors more robust against full tree traversal caused by invalid ray inputs (e.g. INF, NaN, etc)

New Features in Embree 2.6.0

- Added `rtcInterpolate` function to interpolate per vertex attributes
- Added `rtcSetBoundaryMode` function that can be used to select the boundary handling for subdivision surfaces
- Fixed a traversal bug that caused rays with very small ray direction components to miss geometry

- Performance improvements for the robust traversal mode
- Fixed deadlock when calling `rtcCommit` from multiple threads on same scene

New Features in Embree 2.5.1

- On dual socket workstations, the initial BVH build performance almost doubled through a better memory allocation scheme
- Reduced memory usage for subdivision surface objects with crease features
- `rtcCommit` performance is robust against unset "flush to zero" and "denormals are zero" flags. However, enabling these flags in your application is still recommended
- Reduced memory usage for subdivision surfaces with borders and infinitely sharp creases
- Lots of internal cleanups and bug fixes for both Intel® Xeon® and Intel® Xeon Phi™

New Features in Embree 2.5.0

- Improved hierarchy build performance on both Intel Xeon and Intel Xeon Phi
- Vastly improved tessellation cache for ray tracing subdivision surfaces
- Added `rtcGetUserData` API call to query per geometry user pointer set through `rtcSetUserData`
- Added support for memory monitor callback functions to track and limit memory consumption
- Added support for progress monitor callback functions to track build progress and cancel long build operations
- BVH builders can be used to build user defined hierarchies inside the application (see tutorial BVH Builder)
- Switched to TBB as default tasking system on Xeon to get even faster hierarchy build times and better integration for applications that also use TBB
- `rtcCommit` can get called from multiple TBB threads to join the hierarchy build operations

New Features in Embree 2.4

- Support for Catmull Clark subdivision surfaces (triangle/quad base primitives)
- Support for vector displacements on Catmull Clark subdivision surfaces
- Various bug fixes (e.g. 4-byte alignment of vertex buffers works)

New Features in Embree 2.3.3

- BVH builders more robustly handle invalid input data (Intel Xeon processor family)
- Motion blur support for hair geometry (Xeon)
- Improved motion blur performance for triangle geometry (Xeon)
- Improved robust ray tracing mode (Xeon)
- Added `rtcCommitThread` API call for easier integration into existing tasking systems (Xeon and Intel Xeon Phi coprocessor)
- Added support for recording and replaying all `rtcIntersect`/`rtcOccluded` calls (Xeon and Xeon Phi)

New Features in Embree 2.3.2

- Improved mixed AABB/OBB-BVH for hair geometry (Xeon Phi)
- Reduced amount of pre-allocated memory for BVH builders (Xeon Phi)
- New 64 bit Morton code-based BVH builder (Xeon Phi)
- (Enhanced) Morton code-based BVH builders use now tree rotations to improve BVH quality (Xeon Phi)
- Bug fixes (Xeon and Xeon Phi)

New Features in Embree 2.3.1

- High quality BVH mode improves spatial splits which result in up to 30% performance improvement for some scenes (Xeon)
- Compile time enabled intersection filter functions do not reduce performance if no intersection filter is used in the scene (Xeon and Xeon Phi)
- Improved ray tracing performance for hair geometry by >20% on Xeon Phi. BVH for hair geometry requires 20% less memory
- BVH8 for AVX/AVX2 targets improves performance for single ray tracing on Haswell by up to 12% and by up to 5% for hybrid (Xeon)
- Memory conservative BVH for Xeon Phi now uses BVH node quantization to lower memory footprint (requires half the memory footprint of the default BVH)

New Features in Embree 2.3

- Support for ray tracing hair geometry (Xeon and Xeon Phi)
- Catching errors through error callback function
- Faster hybrid traversal (Xeon and Xeon Phi)
- New memory conservative BVH for Xeon Phi
- Faster Morton code-based builder on Xeon
- Faster binned-SAH builder on Xeon Phi
- Lots of code cleanups/simplifications/improvements (Xeon and Xeon Phi)

New Features in Embree 2.2

- Support for motion blur on Xeon Phi
- Support for intersection filter callback functions
- Support for buffer sharing with the application
- Lots of AVX2 optimizations, e.g. ~20% faster 8-wide hybrid traversal
- Experimental support for 8-wide (AVX/AVX2) and 16-wide BVHs (Xeon Phi)

New Features in Embree 2.1

- New future proof API with a strong focus on supporting dynamic scenes
- Lots of optimizations for 8-wide AVX2 (Haswell architecture)
- Automatic runtime code selection for SSE, AVX, and AVX2
- Support for user-defined geometry
- New and improved BVH builders:
  - Fast adaptive Morton code-based builder (without SAH-based top-level rebuild)
  - Both the SAH and Morton code-based builders got faster (Xeon Phi)
  - New variant of the SAH-based builder using triangle pre-splits (Xeon Phi)

Example Performance Numbers for Embree 2.1

BVH rebuild performance (including triangle accel generation, excluding memory allocation) for scenes with 2–12 million triangles:

- Intel® Core™ i7 (Haswell-based CPU, 4 cores @ 3.0 GHz)
  - 7–8 million triangles/s for the SAH-based BVH builder
  - 30–36 million triangles/s for the Morton code-based BVH builder
- Intel® Xeon Phi™ 7120
  - 37–40 million triangles/s for the SAH-based BVH builder
  - 140–160 million triangles/s for the Morton code-based BVH builder

Rendering of the Crown model (`crown.ecs`) with 4 samples per pixel (`-spp 4`):

- Intel® Core™ i7 (Haswell-based CPU, 4 cores CPU @ 3.0 GHz)
  - 1024×1024 resolution: 7.8 million rays per sec
  - 1920×1080 resolution: 9.9 million rays per sec
- Intel® Xeon Phi™ 7120
  - 1024×1024 resolution: 47.1 million rays per sec
  - 1920×1080 resolution: 61.1 million rays per sec

New Features in Embree 2.0

- Support for the Intel® Xeon Phi™ coprocessor platform
- Support for high-performance "packet" kernels on SSE, AVX, and Xeon Phi
- Integration with the Intel® SPMD Program Compiler (ISPC)
- Instantiation and fast BVH reconstruction
- Example photo-realistic rendering engine for both C++ and ISPC

Acknowledgements

This software is based in part on the work of the Independent JPEG Group.

Chapter 2
Installation of Embree

Windows Installer

You can install the 64 bit version of the Embree library using the Windows installer application embree-2.16.0-x64.exe. This will install the 64 bit Embree version by default in `Program Files\Intel\Embree v2.16.0 x64`. To install the 32 bit Embree library use the embree-2.16.0-win32.exe installer. This will install the 32 bit Embree version by default in `Program Files\Intel\Embree v2.16.0 win32` on 32 bit systems and `Program Files (x86)\Intel\Embree v2.16.0 win32` on 64 bit systems.

You have to set the path to the `lib` folder manually to your `PATH` environment variable for applications to find Embree. To compile applications with Embree you also have to set the `Include Directories` path in Visual Studio to the `include` folder of the Embree installation.

To uninstall Embree again open `Programs and Features` by clicking the `Start button`, clicking `Control Panel`, clicking `Programs`, and then clicking `Programs and Features`. Select `Embree 2.16.0` and uninstall it.

Windows ZIP File

Embree is also delivered as a ZIP file for 64 bit embree-2.16.0.x64.windows.zip and 32 bit embree-2.16.0.win32.windows.zip. After unpacking this ZIP file you should set the path to the `lib` folder manually to your `PATH` environment variable for applications to find Embree. To compile applications with Embree you also have to set the `Include Directories` path in Visual Studio to the `include` folder of the Embree installation.

If you plan to ship Embree with your application, best use the Embree version from this ZIP file.

Linux RPMs

Uncompress the 'tar.gz' file embree-2.16.0.x86_64.rpm.tar.gz to obtain the individual RPM files:

```
tar xzf embree-2.16.0.x86_64.rpm.tar.gz
```

To install the Embree using the RPM packages on your Linux system type the following:

```
sudo rpm --install embree-lib-2.16.0-1.x86_64.rpm
sudo rpm --install embree-devel-2.16.0-1.x86_64.rpm
sudo rpm --install embree-examples-2.16.0-1.x86_64.rpm
```

You also have to install the Intel® Threading Building Blocks (TBB) using `yum`:

```
sudo yum install tbb.x86_64 tbb-devel.x86_64
```

or via `apt-get`:

```
sudo apt-get install libtbb-dev
```

Alternatively you can download the latest TBB version from https://www.threadingbuildingblocks.org/download and set the `LD_LIBRARY_PATH` environment variable to point to the TBB library.

Note that the Embree RPMs are linked against the TBB version coming with CentOS. This older TBB version is missing some features required to get optimal build performance and does not support building of scenes lazily during rendering. To get a full featured Embree please install using the tar.gz files, which always ship with the latest TBB version.

Under Linux Embree is installed by default in the `/usr/lib` and `/usr/include` directories. This way applications will find Embree automatically. The Embree tutorials are installed into the `/usr/bin/embree2` folder. Specify the full path to the tutorials to start them.

To uninstall Embree again just execute the following:

```
sudo rpm --erase embree-lib-2.16.0-1.x86_64
sudo rpm --erase embree-devel-2.16.0-1.x86_64
sudo rpm --erase embree-examples-2.16.0-1.x86_64
```

Linux tar.gz files

The Linux version of Embree is also delivered as a tar.gz file embree-2.16.0.x86_64.linux.tar.gz. Unpack this file using `tar` and source the provided `embree-vars.sh` (if you are using the bash shell) or `embree-vars.csh` (if you are using the C shell) to setup the environment properly:

```
tar xzf embree-2.16.0.x64.linux.tar.gz
source embree-2.16.0.x64.linux/embree-vars.sh
```

If you want to ship Embree with your application best use the Embree version provided through the tar.gz file.

We recommend adding a relative RPATH to your application that points to the location Embree (and TBB) can be found, e.g. `$ORIGIN/../lib`.

Mac OS X PKG Installer

To install the Embree library on your Mac OS X system use the provided package installer inside embree-2.16.0.x86_64.dmg. This will install Embree by default into `/opt/local/lib` and `/opt/local/include` directories. The Embree tutorials are installed into the `/Applications/Embree2` folder.

You also have to install the Intel® Threading Building Blocks (TBB) using MacPorts:

```
sudo port install tbb
```

Alternatively you can download the latest TBB version from https://www.threadingbuildingblocks.org/download and set the `DYLD_LIBRARY_PATH` environment variable to point to the TBB library.

To uninstall Embree again execute the uninstaller script `/Applications/Embree2/uninstall.command`.

Mac OS X tar.gz file

The Mac OS X version of Embree is also delivered as a tar.gz file embree-2.16.0.x86_64.macosx.tar.gz. Unpack this file using `tar` and source the provided `embree-vars.sh` (if you are using the bash shell) or `embree-vars.csh` (if you are using the C shell) to setup the environment properly:

```
tar xzf embree-2.16.0.x64.macosx.tar.gz
source embree-2.16.0.x64.macosx/embree-vars.sh
```

If you want to ship Embree with your application please use the Embree library of the provided tar.gz file. The library name of that Embree library is of the form `@rpath/libembree.2.dylib` (and similar also for the included TBB library). This ensures that you can add a relative RPATH to your application that points to the location Embree (and TBB) can be found, e.g. `@loader_path/../lib`.

Linking ISPC applications with Embree

The precompiled Embree library uses the multi-target mode of ISPC. For your ISPC application to properly link against Embree you also have to enable this mode. You can do this by specifying multiple targets when compiling your application with ISPC, e.g.:

```
ispc --target sse2,sse4,avx,avx2 -o code.o code.ispc
```

Chapter 3
Compiling Embree

We recommend to use CMake to build Embree. Do not enable fast-math optimization, these might break Embree.

Linux and Mac OS X

To compile Embree you need a modern C++ compiler that supports C++11. Embree is tested with Intel® Compiler 17.0 (Update 1), Intel® Compiler 16.0 (Update 1), Clang 3.8.0 (supports AVX2), Clang 4.0.0 (supports AVX512) and GCC 5.4.0. If the GCC that comes with your Fedora/Red Hat/CentOS distribution is too old then you can run the provided script `scripts/install_linux_gcc.sh` to locally install a recent GCC into `$HOME/devtools-2`.

Embree supports to use the Intel® Threading Building Blocks (TBB) as tasking system. For performance and flexibility reasons we recommend to use Embree with the Intel® Threading Building Blocks (TBB) and best also use TBB inside your application. Optionally you can disable TBB in Embree through the `EMBREE_TASKING_SYSTEM` CMake variable.

Embree supports the Intel® SPMD Program Compiler (ISPC), which allows straight forward parallelization of an entire renderer. If you do not want to use ISPC then you can disable `EMBREE_ISPC_SUPPORT` in CMake. Otherwise, download and install the ISPC binaries (we have tested ISPC version 1.9.1) from ispc.github.io. After installation, put the path to `ispc` permanently into your `PATH` environment variable or you need to correctly set the `ISPC_EXECUTABLE` variable during CMake configuration.

You additionally have to install CMake 2.8.11 or higher and the developer version of GLUT.

Under Mac OS X, all these dependencies can be installed using MacPorts:

```
sudo port install cmake tbb freeglut
```

Depending on your Linux distribution you can install these dependencies using `yum` or `apt-get`. Some of these packages might already be installed or might have slightly different names.

Type the following to install the dependencies using `yum`:

```
sudo yum install cmake.x86_64
sudo yum install tbb.x86_64 tbb-devel.x86_64
sudo yum install freeglut.x86_64 freeglut-devel.x86_64
sudo yum install libXmu.x86_64 libXi.x86_64
sudo yum install libXmu-devel.x86_64 libXi-devel.x86_64
```

Type the following to install the dependencies using `apt-get`:

```
sudo apt-get install cmake-curses-gui
sudo apt-get install libtbb-dev
sudo apt-get install freeglut3-dev
sudo apt-get install libxmu-dev libxi-dev
```

Finally you can compile Embree using CMake. Create a build directory inside the Embree root directory and execute `ccmake ..` inside this build directory.

```
mkdir build
cd build
ccmake ..
```

Per default CMake will use the compilers specified with the `CC` and `CXX` environment variables. Should you want to use a different compiler, run `cmake` first and set the `CMAKE_CXX_COMPILER` and `CMAKE_C_COMPILER` variables to the desired compiler. For example, to use the Intel® Compiler instead of the default GCC on most Linux machines (`g++` and `gcc`) execute

```
cmake -DCMAKE_CXX_COMPILER=icpc -DCMAKE_C_COMPILER=icc ..
```

Similarly, to use Clang set the variables to `clang++` and `clang`, respectively. Note that the compiler variables cannot be changed anymore after the first run of `cmake` or `ccmake`.

Running `ccmake` will open a dialog where you can perform various configurations as described below in CMake Configuration. After having configured Embree, press c (for configure) and g (for generate) to generate a Makefile and leave the configuration. The code can be compiled by executing make.

```
make
```

The executables will be generated inside the build folder. We recommend to finally install the Embree library and header files on your system. Therefore set the `CMAKE_INSTALL_PREFIX` to `/usr` in cmake and type:

```
sudo make install
```

If you keep the default `CMAKE_INSTALL_PREFIX` of `/usr/local` then you have to make sure the path `/usr/local/lib` is in your `LD_LIBRARY_PATH`.

You can also uninstall Embree again by executing:

```
sudo make uninstall
```

If you cannot install Embree on your system (e.g. when you don't have administrator rights) you need to add embree_root_directory/build to your `LD_LIBRARY_PATH`.

Windows

Embree is tested under Windows using the Visual Studio 2017, Visual Studio 2015 (Update 1) compiler (Win32 and x64), Visual Studio 2013 (Update 5) compiler (Win32 and x64), Intel® Compiler 17.0 (Update 1) (Win32 and x64), Intel® Compiler 16.0 (Update 1) (Win32 and x64), and Clang 3.9 (Win32 and x64). Using the Visual Studio 2015 compiler, Visual Studio 2013 compiler, Intel® Compiler, and Clang you can compile Embree for AVX2. To compile Embree for AVX-512 you have to use the Intel® Compiler.

Embree supports to use the Intel® Threading Building Blocks (TBB) as tasking system. For performance and flexibility reasons we recommend to use Embree with the Intel® Threading Building Blocks (TBB) and best also use TBB inside

your application. Optionally you can disable TBB in Embree through the `EMBREE_TASKING_SYSTEM` CMake variable.

Embree will either find the Intel® Threading Building Blocks (TBB) installation that comes with the Intel® Compiler, or you can install the binary distribution of TBB directly from www.threadingbuildingblocks.org into a folder named tbb into your Embree root directory. You also have to make sure that the libraries tbb.dll and tbb_malloc.dll can be found when executing your Embree applications, e.g. by putting the path to these libraries into your `PATH` environment variable.

Embree supports the Intel® SPMD Program Compiler (ISPC), which allows straight forward parallelization of an entire renderer. If you do not want to use ISPC then you can disable `EMBREE_ISPC_SUPPORT` in CMake. Otherwise, download and install the ISPC binaries (we have tested ISPC version 1.9.1) from ispc.github.io. After installation, put the path to `ispc.exe` permanently into your `PATH` environment variable or you need to correctly set the `ISPC_EXECUTABLE` variable during CMake configuration.

You additionally have to install CMake (version 2.8.11 or higher). Note that you need a native Windows CMake installation, because CMake under Cygwin cannot generate solution files for Visual Studio.

Using the IDE

Run `cmake-gui`, browse to the Embree sources, set the build directory and click Configure. Now you can select the Generator, e.g. "Visual Studio 12 2013" for a 32 bit build or "Visual Studio 12 2013 Win64" for a 64 bit build.

To use a different compile than the Microsoft Visual C++ compiler, you additionally need to specify the proper compiler toolset through the option "Optional toolset to use (-T parameter)". E.g. to use Clang for compilation set the toolset to "LLVM-vs2013", to use the Intel® Compiler 2017 for compilation set the toolset to "Intel C++ Compiler 17.0".

Do not change the toolset manually in a solution file (neither through the project properties dialog, nor through the "Use Intel Compiler" project context menu), as then some compiler specific command line options cannot get set by CMake.

Most configuration parameters described in the CMake Configuration can be set under Windows as well. Finally, click "Generate" to create the Visual Studio solution files.

Table 3.1 – Windows-specific CMake build options for Embree.

Option	Description	Default
CMAKE_CONFIGURATION_TYPE	List of generated configurations.	Debug;Release;RelWithDebInfo
USE_STATIC_RUNTIME	Use the static version of the C/C++ runtime library.	OFF

Use the generated Visual Studio solution file `embree2.sln` to compile the project. To build Embree with support for the AVX2 instruction set you need at least Visual Studio 2013 (Update 4).

We recommend enabling syntax highlighting for the `.ispc` source and `.isph` header files. To do so open Visual Studio, go to Tools ⇒ Options ⇒ Text Editor ⇒ File Extension and add the isph and ispc extension for the "Microsoft Visual C++" editor.

Using the Command Line

Embree can also be configured and built without the IDE using the Visual Studio command prompt:

```
cd path\to\embree
mkdir build
cd build
cmake -G "Visual Studio 12 2013 Win64" ..
cmake --build . --config Release
```

To use to the Intel® Compiler set the proper toolset, e.g. for Intel Compiler 17.0:

```
cmake -G "Visual Studio 12 2013 Win64" -T "Intel C++ Compiler 17.0" ..
cmake --build . --config Release
```

You can also build only some projects with the `--target` switch. Additional parameters after "`--`" will be passed to `msbuild`. For example, to build the Embree library in parallel use

```
cmake --build . --config Release --target embree -- /m
```

CMake Configuration

The default CMake configuration in the configuration dialog should be appropriate for most usages. The following table describes all parameters that can be configured in CMake:

When using the statically compiled Embree library, you have to define ENABLE_STATIC_LIB before including rtcore.h in your application. Further multiple static libraries are generated for the different ISAs selected (e.g. embree2.a, embree2_sse42.a, embree2_avx.a, embree2_avx2.a, embree2_avx512knl.a, embree2_avx512skx.a). You have to link these libraries in increasing ISA order (the order shown in the example).

When selecting ISAs individually, the AVX ISA always has to get enabled when a larger ISA than AVX is enabled, otherwise the code will not compile.

Table 3.2 – CMake build options for Embree.

Option	Description	Default
CMAKE_BUILD_TYPE	Can be used to switch between Debug mode (Debug), Release mode (Release), and Release mode with enabled assertions and debug symbols (RelWithDebInfo).	Release
EMBREE_ISPC_SUPPORT	Enables ISPC support of Embree.	ON
EMBREE_STATIC_LIB	Builds Embree as a static library. When using the statically compiled Embree library, you have to define ENABLE_STATIC_LIB before including rtcore.h in your application.	OFF
EMBREE_IGNORE_CMAKE_CXX_FLAGS	When enabled Embree ignores default CMAKE_CXX_FLAGS.	ON
EMBREE_TUTORIALS	Enables build of Embree tutorials.	ON
EMBREE_BACKFACE_CULLING	Enables backface culling, i.e. only surfaces facing a ray can be hit.	OFF
EMBREE_INTERSECTION_FILTER	Enables the intersection filter feature.	ON
EMBREE_INTERSECTION_FILTER _RESTORE	Restore previous hit when ignoring hits.	ON
EMBREE_RAY_MASK	Enables the ray masking feature.	OFF
EMBREE_RAY_PACKETS	Enables ray packet support.	ON
EMBREE_IGNORE_INVALID_RAYS	Makes code robust against the risk of full-tree traversals caused by invalid rays (e.g. rays containing INF/NaN as origins).	OFF
EMBREE_TASKING_SYSTEM	Chooses between Intel® Threading Building Blocks (TBB) or an internal tasking system (INTERNAL).	TBB
EMBREE_MAX_ISA	Select highest supported ISA (SSE2, SSE4.2, AVX, AVX2, AVX512KNL, AVX512SKX, or NONE). When set to NONE the EMBREE_ISA_* variables can be used to enable ISAs individually.	AVX2
EMBREE_ISA_SSE2	Enables SSE2 when EMBREE_MAX_ISA is set to NONE.	OFF
EMBREE_ISA_SSE42	Enables SSE4.2 when EMBREE_MAX_ISA is set to NONE.	OFF
EMBREE_ISA_AVX	Enables AVX when EMBREE_MAX_ISA is set to NONE.	OFF
EMBREE_ISA_AVX2	Enables AVX2 when EMBREE_MAX_ISA is set to NONE.	OFF
EMBREE_ISA_AVX512KNL	Enables AVX-512 for Xeon Phi when EMBREE_MAX_ISA is set to NONE.	OFF
EMBREE_ISA_AVX512SKX	Enables AVX-512 for Skylake when EMBREE_MAX_ISA is set to NONE.	OFF
EMBREE_GEOMETRY_TRIANGLES	Enables support for triangle geometries.	ON
EMBREE_GEOMETRY_QUADS	Enables support for quad geometries.	ON
EMBREE_GEOMETRY_LINES	Enables support for line geometries.	ON
EMBREE_GEOMETRY_HAIR	Enables support for hair geometries.	ON
EMBREE_GEOMETRY_SUBDIV	Enables support for subdiv geometries.	ON
EMBREE_GEOMETRY_USER	Enables support for user geometries.	ON
EMBREE_NATIVE_SPLINE_BASIS	Specifies the spline basis Embree uses internally for its calculations. Hair and curves using this basis can be rendered directly, others are converted.	BEZIER

Chapter 4
Embree API

The Embree API is a low level ray tracing API that supports defining and committing of geometry and performing ray queries of different types. Static and dynamic scenes are supported, that may contain triangle geometries, quad geometries, line segment geometries, hair geometries, analytic bezier curves, subdivision meshes, instanced geometries, and user defined geometries. For each geometry type multi-segment motion blur is supported, including support for transformation motion blur of instances. Supported ray queries are, finding the closest scene intersection along a ray, and testing a ray segment for any intersection with the scene. Single rays, as well as packets of rays in a struct of array layout can be used for packet sizes of 1, 4, 8, and 16 rays. Using the ray stream interface a stream of an arbitrary number `M` of ray packets of arbitrary size `N` can be processed. Filter callback functions are supported, that get invoked for every intersection encountered during traversal.

The Embree API exists in a C++ and ISPC version. This document describes the C++ version of the API, the ISPC version is almost identical. The only differences are that the ISPC version needs some ISPC specific `uniform` type modifiers, and has special functions that operate on ray packets of the native SIMD size the ISPC code is compiled for.

Embree supports two modes for a scene, the `normal mode` and `stream mode`, which require different ray queries and callbacks to be used. The `normal mode` is the default, but we will switch entirely to the ray `stream mode` in a later release.

The user is supposed to include the `embree2/rtcore.h`, and the `embree2/rtcore_ray.h` file, but none of the other header files. If using the ISPC version of the API, the user should include `embree2/rtcore.isph` and `embree2/rtcore_ray.isph`.

```
#include <embree2/rtcore.h>
#include <embree2/rtcore_ray.h>
```

All API calls carry the prefix `rtc` which stands for **r**ay **t**racing **c**ore. Embree supports a device concept, which allows different components of the application to use the API without interfering with each other. You have to create at least one Embree device through the `rtcNewDevice` call. Before the application exits it should delete all devices by invoking `rtcDeleteDevice`. An application typically creates a single device only, and should create only a small number of devices.

```
RTCDevice device = rtcNewDevice(NULL);
...
rtcDeleteDevice(device);
```

It is strongly recommended to have the `Flush to Zero` and `Denormals are Zero` mode of the MXCSR control and status register enabled for each thread before calling the `rtcIntersect` and `rtcOccluded` functions. Otherwise, under

some circumstances special handling of denormalized floating point numbers can significantly reduce application and Embree performance. When using Embree together with the Intel® Threading Building Blocks, it is sufficient to execute the following code at the beginning of the application main thread (before the creation of the `tbb::task_scheduler_init` object):

```
#include <xmmintrin.h>
#include <pmmintrin.h>
...
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
```

Embree processes some implementation specific configuration from the following locations in the specified order:

1) configuration string passed to the `rtcNewDevice` function
2) `.embree2` file in the application folder
3) `.embree2` file in the home folder

Settings performed later overwrite previous settings. This way the configuration for the application can be changed globally (either through the `rtcNewDevice` call or through the `.embree2` file in the application folder) and each user has the option to modify the configuration to fit its needs. Configuration files can be ignored by the application by passing `ignore_config_files=1` to `rtcNewDevice`.

API calls that access geometries are only thread safe as long as different geometries are accessed. Accesses to one geometry have to get sequenced by the application. All other API calls are thread safe. The API calls are re-entrant, it is thus safe to trace new rays and create new geometry when intersecting a user defined object.

Each user thread has its own error flag per device. If an error occurs when invoking some API function, this flag is set to an error code if it stores no previous error. The `rtcDeviceGetError` function reads and returns the currently stored error and clears the error flag again.

Possible error codes returned by `rtcDeviceGetError` are:

Table 4.1 – Return values of `rtcDeviceGetError`.

Error Code	Description
RTC_NO_ERROR	No error occurred.
RTC_UNKNOWN_ERROR	An unknown error has occurred.
RTC_INVALID_ARGUMENT	An invalid argument was specified.
RTC_INVALID_OPERATION	The operation is not allowed for the specified object.
RTC_OUT_OF_MEMORY	There is not enough memory left to complete the operation.
RTC_UNSUPPORTED_CPU	The CPU is not supported as it does not support SSE2.
RTC_CANCELLED	The operation got cancelled by an Memory Monitor Callback or Progress Monitor Callback function.

When the device construction fails `rtcNewDevice` returns `NULL` as device. To detect the error code of a such a failed device construction pass `NULL` as device to the `rtcDeviceGetError` function. For all other invokations of `rtcDeviceGetError` a proper device pointer has to get specified.

Using the `rtcDeviceSetErrorFunction2` call, it is also possible to set a callback function that is called whenever an error occurs for a device.

```
typedef void (*RTCErrorFunc2)(void* userPtr, const RTCError code, const char* str);
void rtcDeviceSetErrorFunction2(RTCDevice device, RTCErrorFunc2 func, void* userPtr);
```

When invoked, the registred callback function gets passed a user defined pointer `userPtr`, the error code `code`, as well as some string `str` that describes the error further. Passing `NULL` as function pointer to `rtcDeviceSetErrorFunction2` disables the set callback function again. The previously described error flags are also set if an error callback function is present.

Scene

A scene is a container for a set of geometries of potentially different types. A scene is created using the `rtcDeviceNewScene` function call, and destroyed using the `rtcDeleteScene` function call. Two types of scenes are supported, dynamic and static scenes. Different flags specify the type of scene to create and the type of ray query operations that can later be performed on the scene. The following example creates a scene that supports dynamic updates and the single ray `rtcIntersect` and `rtcOccluded` calls.

```
RTCScene scene = rtcDeviceNewScene(device, RTC_SCENE_DYNAMIC, RTC_INTERSECT1);
...
rtcDeleteScene(scene);
```

Using the following scene flags the user can select between creating a static or dynamic scene.

Scene Flag	Description
RTC_SCENE_STATIC	Scene is optimized for static geometry.
RTC_SCENE_DYNAMIC	Scene is optimized for dynamic geometry.

Table 4.2 – Dynamic type flags for `rtcDeviceNewScene`.

A dynamic scene is created by invoking `rtcDeviceNewScene` with the `RTC_SCENE_DYNAMIC` flag. Different geometries can now be created inside that scene. Geometries are enabled by default. Once the scene geometry is specified, an `rtcCommit` call will finish the scene description and trigger building of internal data structures. After the `rtcCommit` call it is safe to perform ray queries of the type specified at scene construction time. Geometries can get disabled (`rtcDisable` call), enabled again (`rtcEnable` call), and deleted (`rtcDeleteGeometry` call). Geometries can also get modified, including their vertex and index arrays. After the modification of some geometry, `rtcUpdate` or `rtcUpdateBuffer` has to get called for that geometry to specify which buffers got modified. Each modified buffer can be specified separately using the `rtcUpdateBuffer` function. In contrast the `rtcUpdate` function simply tags each buffer of some geometry as modified. If geometries got enabled, disabled, deleted, or modified an `rtcCommit` call has to get invoked before performing any ray queries for the scene, otherwise the effect of the ray query is undefined. During an `rtcCommit` call modifications to the scene are not allowed.

A static scene is created by the `rtcDeviceNewScene` call with the `RTC_SCENE_STATIC` flag. Geometries can only get created, enabled, disabled and modified until the first `rtcCommit` call. After the `rtcCommit` call, each access to any geometry of that static scene is invalid. Geometries that got created inside a static scene can only get deleted by deleting the entire scene.

The modification of geometry, building of hierarchies using `rtcCommit`, and tracing of rays have always to happen separately, never at the same time.

Embree silently ignores primitives that would cause numerical issues, e.g. primitives containing NaNs, INFs, or values greater than 1.844E18f.

The following flags can be used to tune the used acceleration structure. These flags are only hints and may be ignored by the implementation.

Table 4.3 – Acceleration structure flags for `rtcDeviceNewScene`.

Scene Flag	Description
RTC_SCENE_COMPACT	Creates a compact data structure and avoids algorithms that consume much memory.
RTC_SCENE_COHERENT	Optimize for coherent rays (e.g. primary rays).
RTC_SCENE_INCOHERENT	Optimize for in-coherent rays (e.g. diffuse reflection rays).
RTC_SCENE_HIGH_QUALITY	Build higher quality spatial data structures.

The following flags can be used to tune the traversal algorithm that is used by Embree. These flags are only hints and may be ignored by the implementation.

Scene Flag	Description
RTC_SCENE_ROBUST	Avoid optimizations that reduce arithmetic accuracy.

Table 4.4 – Traversal algorithm flags for `rtcDeviceNewScene`.

The second argument of the `rtcDeviceNewScene` function are algorithm flags, that allow to specify which ray queries are required by the application. Calling a ray query API function for a scene that is different to the ones specified at scene creation time is not allowed. Further, the application should only pass ray query requirements that are really needed, to give Embree most freedom in choosing the best algorithm. E.g. in case Embree implements no packet traversers for some highly optimized data structure for single rays, then this data structure cannot be used if the user enables any ray packet query.

Table 4.5 – Enabled algorithm flags for `rtcDeviceNewScene`.

Algorithm Flag	Description
RTC_INTERSECT1	Enables the `rtcIntersect` and `rtcOccluded` functions (single ray interface) for this scene.
RTC_INTERSECT4	Enables the `rtcIntersect4` and `rtcOccluded4` functions (4-wide packet interface) for this scene.
RTC_INTERSECT8	Enables the `rtcIntersect8` and `rtcOccluded8` functions (8-wide packet interface) for this scene.
RTC_INTERSECT16	Enables the `rtcIntersect16` and `rtcOccluded16` functions (16-wide packet interface) for this scene.
RTC_INTERSECT_STREAM	Enables the `rtcIntersect1M`, `rtcOccluded1M`, `rtcIntersect1Mp`, `rtcOccluded1Mp`, `rtcIntersectNM`, `rtcOccludedNM`, `rtcIntersectNp`, and `rtcOccludedNp` functions for this scene.
RTC_INTERPOLATE	Enables the `rtcInterpolate` and `rtcInterpolateN` interpolation functions.

Embree supports two modes for a scene, the `normal mode` and `stream mode`. These modes mainly differ in the kind of callbacks invoked and how rays are extended with user data. The normal mode is enabled by default, the ray stream mode can be enabled using the `RTC_INTERSECT_STREAM` algorithm flag for a scene. Only in ray stream mode, the stream API functions `rtcIntersect1M`, `rtcIntersect1Mp`, `rtcIntersectNM`, and `rtcIntersectNp` as well as their occlusion variants can be used.

The scene bounding box can get read by the function `rtcGetBounds(RTCScene scene, RTCBounds& bounds_o)`. This function will write the AABB of the scene to `bounds_o`. Time varying bounds can be obtained usin the `rtcGetLinearBounds(RTCScene scene, RTCBounds* bounds_o)` function. This function will write two AABBs to `bounds_o`. Linearly interpolating these bounds to a specific time `t` yields bounds that bound the geometry at that time. Invoking these functions is only valid when all scene changes got committed using `rtcCommit`.

Geometries

Geometries are always contained in the scene they are created in. Each geometry is assigned an integer ID at creation time, which is unique for that scene. The current version of the API supports triangle meshes (`rtcNewTriangleMesh2`), quad meshes (`rtcNewQuadMesh2`), Catmull-Clark subdivision surfaces (`rtcNewSubdivisionMesh2`), curve geometries (`rtcNewBezierCurveGeometry2`), hair geometries (`rtcNewBezierHairGeometry2`), single level instances of other scenes (`rtcNewInstance3`), and user defined geometries (`rtcNewUserGeometry3`). The API is designed in a way that easily allows adding new geometry types in later releases.

The application can manage geometry IDs itself, or let Embree allocate geometry IDs. Therefore all geometry creation functions have a `geomID` parameter. This parameter can be set to `RTC_INVALID_GEOMETRY_ID` to let Embree allocate a geometry ID (default) or to some geometry ID allocated by the application.

If the application allocates a geometry ID, then this geometry ID has to be unused in the scene, otherwise the creation of the geometry will fail. Further, the geometry IDs allocated by the application should be compact, as Embree internally created a vector which size is equal to the largest geometry ID used. Creating very large geometry IDs for small scenes would thus cause a memory consumption and performance overhead.

If Embree allocates a geometry ID then the following properties hold. For dynamic scenes, all IDs are assigned sequentially, starting from 0, as long as no geometry got deleted. If geometries got deleted, the implementation will reuse IDs later on in an implementation dependent way. Consequently sequential assignment is no longer guaranteed, but a compact range of IDs. These rules allow the application to manage a dynamic array to efficiently map from geometry IDs to its own geometry representation. For static scenes, geometry IDs are assigned sequentially starting at 0. This allows the application to use a fixed size array to map from geometry IDs to its own geometry representation.

Alternatively the application can also use the `void rtcSetUserData (RTCScene scene, unsigned geomID, void* ptr)` function to set a user data pointer `ptr` to its own geometry representation, and later read out this pointer again using the `void* rtcGetUserData (RTCScene scene, unsigned geomID)` function.

The following geometry flags can be specified at construction time of geometries:

Triangle Meshes

Triangle meshes are created using the `rtcNewTriangleMesh2` function call, and potentially deleted using the `rtcDeleteGeometry` function call.

The number of triangles, number of vertices, and optionally the number of time steps for multi-segment motion blur have to get specified at construction time of the mesh. The user can also specify additional flags that choose the strategy to handle that mesh in dynamic scenes. The following example demonstrates how to create a triangle mesh without motion blur:

Table 4.6 – Flags for the creation of new geometries.

Geometry Flag	Description
RTC_GEOMETRY_STATIC	The geometry is considered static and should get modified rarely by the application. This flag has to get used in static scenes.
RTC_GEOMETRY_DEFORMABLE	The geometry is considered to deform in a coherent way, e.g. a skinned character. The connectivity of the geometry has to stay constant, thus modifying the index array is not allowed. The implementation is free to choose a BVH refitting approach for handling meshes tagged with that flag.
RTC_GEOMETRY_DYNAMIC	The geometry is considered highly dynamic and changes frequently, possibly in an unstructured way. Embree will rebuild data structures from scratch for this type of geometry.

```
unsigned geomID = rtcNewTriangleMesh2(scene, geomFlags,
                                      numTriangles, numVertices, 1);
```

The triangle indices can be set by mapping and writing to the index buffer (`RTC_INDEX_BUFFER`) and the triangle vertices can be set by mapping and writing into the vertex buffer (`RTC_VERTEX_BUFFER`). The index buffer contains an array of three 32 bit indices, while the vertex buffer contains an array of three float values. The vertex buffer can be at most 16GB large. When the vertex buffer is managed internally the stride between vertices is 16 bytes. For multi segment motion blur, for each time step a vertex buffer has to be specified, and all these buffers have to have the same stride. All buffers have to get unmapped before an `rtcCommit` call to the scene.

```
struct Vertex   { float x, y, z, a; };
struct Triangle { int v0, v1, v2; };

Vertex* vertices = (Vertex*) rtcMapBuffer(scene, geomID, RTC_VERTEX_BUFFER);
// fill vertices here
rtcUnmapBuffer(scene, geomID, RTC_VERTEX_BUFFER);

Triangle* triangles = (Triangle*) rtcMapBuffer(scene, geomID, RTC_INDEX_BUFFER);
// fill triangle indices here
rtcUnmapBuffer(scene, geomID, RTC_INDEX_BUFFER);
```

Also see tutorial Triangle Geometry for an example of how to create triangle meshes.

The parametrization of a triangle uses the first vertex `p0` as base point, and the vector `p1 - p0` as u-direction and `p2 - p0` as v-direction. The following picture additionally illustrates the direction the geometry normal is pointing into.

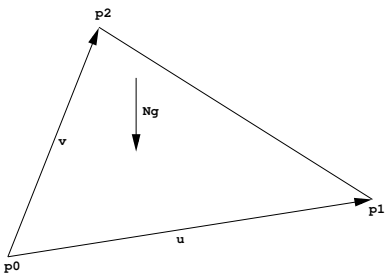

Figure 4.1

Some texture coordinates `t0,t1,t2` can be linearly interpolated over the triangle the following way:

```
t_uv = (1-u-v)*t0 + u*t1 + v*t2
```

Quad Meshes

Quad meshes are created using the `rtcNewQuadMesh2` function call, and potentially deleted using the `rtcDeleteGeometry` function call.

The number of quads, number of vertices, and optionally the number of time steps for multi-segment motion blur have to get specified at construction time of the mesh. The user can also specify additional flags that choose the strategy to handle that mesh in dynamic scenes. The following example demonstrates how to create a quad mesh without motion blur:

```
unsigned geomID = rtcNewQuadMesh2(scene, geomFlags,
                                  numQuads, numVertices, 1);
```

The quad indices can be set by mapping and writing to the index buffer (`RTC_INDEX_BUFFER`) and the quad vertices can be set by mapping and writing into the vertex buffer (`RTC_VERTEX_BUFFER`). The index buffer contains an array of four 32 bit indices, while the vertex buffer contains an array of three float values. The vertex buffer can be at most 16GB large. When the vertex buffer is managed internally the stride between vertices is 16 bytes. For multi segment motion blur, for each time step a vertex buffer has to be specified, and all these buffers have to have the same stride. All buffers have to get unmapped before an `rtcCommit` call to the scene.

```
struct Vertex { float x, y, z, a; };
struct Quad   { int v0, v1, v2, v3; };

Vertex* vertices = (Vertex*) rtcMapBuffer(scene, geomID, RTC_VERTEX_BUFFER);
// fill vertices here
rtcUnmapBuffer(scene, geomID, RTC_VERTEX_BUFFER);

Quad* quads = (Quad*) rtcMapBuffer(scene, geomID, RTC_INDEX_BUFFER);
// fill quad indices here
rtcUnmapBuffer(scene, geomID, RTC_INDEX_BUFFER);
```

A quad is internally handled as a pair of two triangles `v0,v1,v3` and `v2,v3,v1`, with the u'/v' coordinates of the second triangle corrected by `u = 1-u'` and `v = 1-v'` to produce a quad parametrization where u and v go from 0 to 1.

To encode a triangle as a quad just replicate the last triangle vertex (`v0,v1,v2 -> v0,v1,v2,v2`). This way the quad mesh can be used to represent a mixed mesh which contains triangles and quads.

Subdivision Surfaces

Catmull-Clark subdivision surfaces for meshes consisting of faces of up to 15 vertices (e.g. triangles, quadrilateral, pentagons, etc.) are supported, including support for edge creases, vertex creases, holes, non-manifold geometry, and face-varying interpolation.

A subdivision surface is created using the `rtcNewSubdivisionMesh2` function call, and deleted again using the `rtcDeleteGeometry` function call.

```
unsigned rtcNewSubdivisionMesh2(RTCScene scene,
                                RTCGeometryFlags flags,
                                size_t numFaces,
                                size_t numEdges,
                                size_t numVertices,
                                size_t numEdgeCreases,
                                size_t numVertexCreases,
                                size_t numCorners,
```

```
                                    size_t numHoles,
                                    size_t numTimeSteps,
                                    unsigned int geomID);
```

The number of faces (numFaces), edges/indices (numEdges), vertices (numVertices), edge creases (numEdgeCreases), vertex creases (numVertexCreases), holes (numHoles), and time steps (numTimeSteps) have to get specified at construction time.

The following buffers have to get setup by the application: the face buffer (RTC_FACE_BUFFER) contains the number edges/indices (3 to 15) of each of the numFaces faces, the index buffer (RTC_INDEX_BUFFER) contains multiple (3 to 15) 32 bit vertex indices for each face and numEdges indices in total, the vertex buffer (RTC_VERTEX_BUFFER) stores numVertices vertices as single precision x, y, z floating point coordinates aligned to 16 bytes. The value of the 4th float used for alignment can be arbitrary.

Optionally the application may fill additional index buffers if multiple topologies are required for face-varying interpolation. The standard vertex buffers RTC_VERTEX_BUFFER are always bound to the geometry topology (topology 0) thus use RTC_INDEX_BUFFER0. Data interpolation may use different topologies as described later.

Optionally, the application can setup the hole buffer (RTC_HOLE_BUFFER) with numHoles many 32 bit indices of faces that should be considered non-existing in all topologies.

Optionally, the application can fill the level buffer (RTC_LEVEL_BUFFER) with a tessellation rate for each or the edges of each face, making a total of numEdges values. The tessellation level is a positive floating point value, that specifies how many quads along the edge should get generated during tessellation. If no level buffer is specified a level of 1 is used. The maximally supported edge level is 4096 and larger levels get clamped to that value. Note that some edge may be shared between (typically 2) faces. To guarantee a watertight tessellation, the level of these shared edges has to be exactly identical. A uniform tessellation rate for an entire subdivision mesh can be set by using the `rtcSetTessellationRate(RTCScene scene, unsigned geomID, float rate)` function. The existance of a level buffer has preference over the uniform tessellation rate.

Optionally, the application can fill the sparse edge crease buffers to make some edges appear sharper. The edge crease index buffer (RTC_EDGE_CREASE_INDEX_BUFFER) contains numEdgeCreases many pairs of 32 bit vertex indices that specify unoriented edges in the geometry topology. The edge crease weight buffer (RTC_EDGE_CREASE_WEIGHT_BUFFER) stores for each of theses crease edges a positive floating point weight. The larger this weight, the sharper the edge. Specifying a weight of infinity is supported and marks an edge as infinitely sharp. Storing an edge multiple times with the same crease weight is allowed, but has lower performance. Storing an edge multiple times with different crease weights results in undefined behavior. For a stored edge (i,j), the reverse direction edges (j,i) does not have to get stored, as both are considered the same edge. Edge crease features are specified for the geomtetry topology, but copied to all other topologies automatically.

Optionally, the application can fill the sparse vertex crease buffers to make some vertices appear sharper. The vertex crease index buffer (RTC_VERTEX_CREASE_INDEX_BUFFER), contains numVertexCreases many 32 bit vertex indices to specify a set of vertices from the geometry topology. The vertex crease weight buffer (RTC_VERTEX_CREASE_WEIGHT_BUFFER) specifies for each of these vertices a positive floating point weight. The larger this weight, the sharper the vertex. Specifying a weight of infinity is supported and makes the vertex infinitely sharp. Storing a vertex multiple times with the same crease weight is allowed, but has lower performance. Storing a vertex multiple times with different crease weights results in undefined behavior. Vertex crease features are

specified for the geomtetry topology, but copied to all other topologies automatically.

Faces with 3 to 15 vertices are supported (triangles, quadrilateral, pentagons, etc).

The user can also specify a geometry mask and additional flags that choose the strategy to handle that subdivision mesh in dynamic scenes.

The implementation of subdivision surfaces uses an internal software cache, which can get configured to some desired size (see Configuring Embree).

Parametrization

The parametrization of a regular quadrilateral uses the first vertex `p0` as base point, and the vector `p1 - p0` as u-direction and `p3 - p0` as v-direction. The following picture additionally illustrates the direction the geometry normal is pointing into.

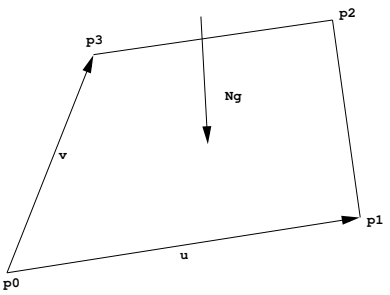

Figure 4.2

Some texture coordinates `t0,t1,t2,t3` can be bi-linearly interpolated over the quadrilateral the following way:

```
t_uv = (1-v)((1-u)*t0 + u*t1) + v*((1-u)*t3 + u*t2)
```

The parametrization for all other face types where the number of vertices is not equal to 4, have a special parametrization where the n'th quadrilateral (that would be obtained by a single subdivision step) is encoded in the higher order bits of the UV coordinates and the local hit location inside this quadrilateral in the lower order bits. The following piece of code extracts the sub-patch ID i and UVs of this subpatch:

```
const unsigned l = floorf(4.0f*U);
const unsigned h = floorf(4.0f*V);
const unsigned i = 4*h+l;
const float u = 2.0f*fracf(4.0f*U);
const float v = 2.0f*fracf(4.0f*V);
```

To smoothly interpolate texture coordinates over the subdivision surface we recommend using the `rtcInterpolate2` function, which will apply the standard subdivision rules for interpolation and automatically take care of the special UV encoding for non-quadrilaterals.

Face-Varing Data

Face-varying interpolation is supported through multiple topologies per subdivision mesh and binding such topologies to user vertex buffers to interpolate. This way texture coordinates may use a different topology with additional boundaries to construct separate UV regions inside one subdivision mesh.

Each such topology consists of an index buffer and subdivision mode. Up to 16 topologies are supported, with corresponding index buffers `RTC_INDEX_BUFFER0+i`, with i in the range 0 to 15.

Each of the 16 supported user vertex buffers `RTC_USER_VERTEX_BUFFER0+j` (j in the range 0 to 15) can be assigned to some topology using the `rtcSetIndexBuffer` call:

```
void rtcSetIndexBuffer(RTCScene scene, unsigned geomID,
                       RTCBufferType vertexBuffer, RTCBufferType indexBuffer);
```

The face buffer (`RTC_FACE_BUFFER`) is shared between all topologies, which means that the n'th primitive always has the same number of vertices (e.g. being a triangle or a quad) for each topology. However, the indices of the topologies themselves may be different.

Subdivision Mode

The subdivision modes can be used to force linear interpolation for some parts of the subdivision mesh.

Table 4.7 – Subdivision modes supported by Embree.

Boundary Mode	Description
RTC_SUBDIV_NO_BOUNDARY	Boundary patches are ignored. This way each rendered patch has a full set of control vertices.
RTC_SUBDIV_SMOOTH_BOUNDARY	The sequence of boundary control points are used to generate a smooth B-spline boundary curve (default mode).
RTC_SUBDIV_PIN_CORNERS	Corner vertices are pinned to their location during subdivision.
RTC_SUBDIV_PIN_BOUNDARY	All vertices at the border are pinned to their location during subdivision. This way the boundary is interpolated linearly.
RTC_SUBDIV_PIN_ALL	All vertices at the border are binned to their location during subdivision. This way all patches are linearly interpolated.

These modes can be set to each topology separately using the `rtcSetSubdivisionMode` API call with the following signature:

```
void rtcSetSubdivisionMode(RTCScene scene, unsigned geomID,
                           unsigned topologyID, RTCSubdivisionMode mode);
```

These modes are typically used to interpolate face-varying data properly. E.g. the topology used to interpolate texture coordinaces are typically assigned the `RTC_SUBDIV_PIN_BOUNDARY` mode, to also map texels at the border of the texture to the mesh.

Also see tutorial Subdivision Geometry for an example of how to create subdivision surfaces.

Line Segment Hair Geometry

Line segments are supported to render hair geometry. A line segment consists of a start and end point, and start and end radius. Individual line segments are considered to be subpixel sized which allows the implementation to approximate the intersection calculation. This in particular means that zooming onto one line segment might show geometric artifacts.

Line segments are created using the `rtcNewLineSegments2` function call, and potentially deleted using the `rtcDeleteGeometry` function call.

The number of line segments, the number of vertices, and optionally the number of time steps for multi-segment motion blur have to get specified at construction time of the line segment geometry.

The segment indices can be set by mapping and writing to the index buffer (RTC_INDEX_BUFFER) and the vertices can be set by mapping and writing into the vertex buffer (RTC_VERTEX_BUFFER). In case of motion blur, the vertex buffers (RTC_VERTEX_BUFFER0+t) have to get filled for each time step t.

The index buffer contains an array of 32 bit indices pointing to the ID of the first of two vertices, while the vertex buffer stores all control points in the form of a single precision position and radius stored in x, y, z, r order in memory. The radii have to be greater or equal zero. All buffers have to get unmapped before an rtcCommit call to the scene.

The intersection with the line segment primitive stores the parametric hit location along the line segment as u-coordinate (range $[0, 1]$; v is always set to zero). The geometry normal Ng is filled with the the tangent, i.e. the vector from start to end vertex.

Like for triangle meshes, the user can also specify a geometry mask and additional flags that choose the strategy to handle that mesh in dynamic scenes.

The following example demonstrates how to create some line segment geometry:

```
unsigned geomID = rtcNewLineSegments2(scene, geomFlags, numCurves,
                                      numVertices, 1);

struct Vertex { float x, y, z, r; };

Vertex* vertices = (Vertex*) rtcMapBuffer(scene, geomID, RTC_VERTEX_BUFFER);
// fill vertices here
rtcUnmapBuffer(scene, geomID, RTC_VERTEX_BUFFER);

int* curves = (int*) rtcMapBuffer(scene, geomID, RTC_INDEX_BUFFER);
// fill indices here
rtcUnmapBuffer(scene, geomID, RTC_INDEX_BUFFER);
```

Spline Hair Geometry

Hair geometries are supported, which consist of multiple hairs represented as cubic spline curves with varying radius per control point. As spline basis we currently support Bézier splines and B-splines. Individual hairs are considered to be subpixel sized which allows the implementation to approximate the intersection calculation. This in particular means that zooming onto one hair might show geometric artifacts.

Hair geometries are created using the rtcNewBezierHairGeometry2 or rtcNewBSplineHairGeometry2 function call, and potentially deleted using the rtcDeleteGeometry function call.

The number of hair curves, the number of vertices, and optionally the number of time steps for multi-segment motion blur have to get specified at construction time of the hair geometry.

The curve indices can be set by mapping and writing to the index buffer (RTC_INDEX_BUFFER) and the control vertices can be set by mapping and writing into the vertex buffer (RTC_VERTEX_BUFFER). In case of motion blur, the vertex buffers RTC_VERTEX_BUFFER0+t have to get filled for each time step.

The index buffer contains an array of 32 bit indices pointing to the ID of the first of four control vertices, while the vertex buffer stores all control points in the form of a single precision position and radius stored in x, y, z, r order in memory. The hair radii have to be greater or equal zero. All buffers have to get unmapped before an rtcCommit call to the scene.

The intersection with the hair primitive stores the parametric hit location along the hair as u-coordinate (range 0 to +1), and the normalized distance as

the v-coordinate (range -1 to +1). The geometry normal Ng is filled with the the tangent of the bezier curve at the hit location on the curve (dPdu).

The implementation may choose to subdivide the Bézier curve into multiple cylinders-like primitives. The number of cylinders the curve gets subdivided into can be specified per hair geometry through the `rtcSetTessellationRate(RTCScene scene, unsigned geomID, float rate)` function. By default the tessellation rate for hair curves is 4.

Like for triangle meshes, the user can also specify a geometry mask and additional flags that choose the strategy to handle that mesh in dynamic scenes.

The following example demonstrates how to create some hair geometry:

```
unsigned geomID = rtcNewBezierHairGeometry2(scene, geomFlags, numCurves, numVertices);

struct Vertex { float x, y, z, r; };

Vertex* vertices = (Vertex*) rtcMapBuffer(scene, geomID, RTC_VERTEX_BUFFER);
// fill vertices here
rtcUnmapBuffer(scene, geomID, RTC_VERTEX_BUFFER);

int* curves = (int*) rtcMapBuffer(scene, geomID, RTC_INDEX_BUFFER);
// fill indices here
rtcUnmapBuffer(scene, geomID, RTC_INDEX_BUFFER);
```

Also see tutorial Hair for an example of how to create and use hair geometry.

Spline Curve Geometry

The spline curve geometry consists of multiple cubic spline curves with varying radius per control point. As spline basis we currently support Bézier splines and B-splines. The cuve surface is defined as the sweep surface of sweeping a varying radius circle tangential along the Bézier curve. As a limitation, the radius of the curve has to be smaller than the curvature radius of the Bézier curve at each location on the curve. In contrast to hair geometry, the curve geometry is rendered properly even in closeups.

Curve geometries are created using the `rtcNewBezierCurveGeometry2` or `rtcNewBSplineCurveGeometry2` function call, and potentially deleted using the `rtcDeleteGeometry` function call.

The number of Bézier curves, the number of vertices, and optionally the number of time steps for multi-segment motion blur have to get specified at construction time of the curve geometry.

The curve indices can be set by mapping and writing to the index buffer (`RTC_INDEX_BUFFER`) and the control vertices can be set by mapping and writing into the vertex buffer (`RTC_VERTEX_BUFFER`). In case of motion blur, the vertex buffers `RTC_VERTEX_BUFFER0+t` have to get filled for each time step.

The index buffer contains an array of 32 bit indices pointing to the ID of the first of four control vertices, while the vertex buffer stores all control points in the form of a single precision position and radius stored in `x`, `y`, `z`, `r` order in memory. The curve radii have to be greater or equal zero. All buffers have to get unmapped before an `rtcCommit` call to the scene.

Like for triangle meshes, the user can also specify a geometry mask and additional flags that choose the strategy to handle the curves in dynamic scenes.

Also see tutorial Curves for an example of how to create and use Bézier curve geometries.

User Defined Geometry

User defined geometries make it possible to extend Embree with arbitrary types of user defined primitives. This is achieved by introducing arrays of user primitives as a special geometry type.

User geometries are created using the `rtcNewUserGeometry3` function call, and potentially deleted using the `rtcDeleteGeometry` function call.

When creating a user defined geometry, the user has to set a data pointer, a bounding function closure (function and user pointer) as well as user defined intersect and occluded callback function pointers. The bounding function is used to query the bounds of all timesteps of a user primitive, while the intersect and occluded callback functions are called to intersect the primitive with a ray.

The bounding function to register has the following signature

```
typedef void (*RTCBoundsFunc3)(void* userPtr, void* geomUserPtr, size_t id, size_t timeStep, RTCBounds& bour
```

and can be registered using the `rtcSetBoundsFunction2` API function:

```
rtcSetBoundsFunction3(scene, geomID, userBoundsFunction, userPtr);
```

When the bounding callback is called, it is passed a user defined pointer specified at registration time of the bounds function (`userPtr` parameter), the per geometry user data pointer (`geomUserPtr` parameter), the ID of the primitive to calculate the bounds for (`id` parameter), the time step at which to calculate the bounds (`timeStep` parameter) and a memory location to write the calculated bound to (`bounds_o` parameter).

The signature of supported user defined intersect and occluded function in normal mode is as follows:

```
typedef void (*RTCIntersectFunc  ) (                  void* userDataPtr, RTCRay& ray, size_t item);
typedef void (*RTCIntersectFunc4 ) (const void* valid, void* userDataPtr, RTCRay4& ray, size_t item);
typedef void (*RTCIntersectFunc8 ) (const void* valid, void* userDataPtr, RTCRay8& ray, size_t item);
typedef void (*RTCIntersectFunc16) (const void* valid, void* userDataPtr, RTCRay16& ray, size_t item);
```

The `RTCIntersectFunc` callback function operates on single rays and gets passed the user data pointer of the user geometry (`userDataPtr` parameter), the ray to intersect (`ray` parameter), and the ID of the primitive to intersect (`item` parameter). The `RTCIntersectFunc4/8/16` callback functions operate on ray packets of size 4, 8 and 16 and additionally get an integer valid mask as input (`valid` parameter). The callback functions should not modify any ray that is disabled by that valid mask.

In stream mode the following callback function has to get used:

```
typedef void (*RTCIntersectFuncN ) (const int*  valid, void* userDataPtr, const RTCIntersectContext* context
typedef void (*RTCIntersectFunc1Mp)(                  void* userDataPtr, const RTCIntersectContext* context
```

The `RTCIntersectFuncN` callback function supports ray packets of arbitrary size `N`. The `RTCIntersectFunc1Mp` callback function get an array of `M` pointers to single rays as input.

The user intersect function should return without modifying the ray structure if the user geometry is missed. Whereas, if an intersection of the user primitive with the ray segment was found, the intersect function has to update the hit information of the ray (`tfar`, `u`, `v`, `Ng`, `geomID`, `primID` components).

The user occluded function should also return without modifying the ray structure if the user geometry is missed. If the geometry is hit, it should set the `geomID` member of the ray to 0.

When performing ray queries using the `rtcIntersect` and `rtcOccluded` function, callbacks of type `RTCIntersectFunc` are invoked for user geometries.

Consequently, an application only operating on single rays only has to provide the single ray intersect and occluded callbacks. Similar when calling the `rtcIntersect4/8/16` and `rtcOccluded4/8/16` functions, the `RTCIntersectFunc4/8/16` callbacks of matching packet size and type are called.

If ray stream mode is enabled for the scene only the `RTCIntersectFuncN` and `RTCIntersectFunc1Mp` callback can be used. In this case specifying an `RTCIntersectFuncN` callback is mandatory and the `RTCIntersectFunc1Mp` callback is optional. Trying to set a different type of user callback function results in an error.

The following example illustrates creating an array with two user geometries:

```
int numTimeSteps = 2;
struct UserObject { ... };

void userBoundsFunction(void* userPtr, UserObject* userGeomPtr, size_t i, size_t t, RTCBounds& bounds)
{
    bounds = <bounds of userGeomPtr[i] at time t>;
}

void userIntersectFunction(UserObject* userGeomPtr, RTCRay& ray, size_t i)
{
  if (<ray misses userGeomPtr[i] at time ray.time>)
    return;
  <update ray hit information>;
}

void userOccludedFunction(UserObject* userGeomPtr, RTCRay& ray, size_t i)
{
  if (<ray misses userGeomPtr[i] at time ray.time>)
    return;
  geomID = 0;
}

...

UserObject* userGeomPtr = new UserObject[2];
userGeomPtr[0] = ...
userGeomPtr[1] = ...
unsigned geomID = rtcNewUserGeometry3(scene, 2, numTimeSteps);
rtcSetUserData(scene, geomID, userGeomPtr);
rtcSetBoundsFunction3(scene, geomID, userBoundsFunction, userPtr);
rtcSetIntersectFunction(scene, geomID, userIntersectFunction);
rtcSetOccludedFunction(scene, geomID, userOccludedFunction);
```

See tutorial User Geometry for an example of how to use the user defined geometries.

Instances

Embree supports instancing of scenes inside another scene by some transformation. As the instanced scene is stored only a single time, even if instanced to multiple locations, this feature can be used to create very large scenes. Only single level instancing is supported by Embree natively, however, multi-level instancing can be implemented through user geometries.

Instances are created using the `rtcNewInstance3 (RTCScene target, RTCScene source, size_t numTimeSteps)` function call, and potentially deleted using the `rtcDeleteGeometry` function call. To instantiate a scene, one first has

to generate the scene B to instantiate. Now one can add an instance of this scene inside a scene A the following way:

```
unsigned instID = rtcNewInstance3(sceneA, sceneB, 1);
rtcSetTransform2(sceneA, instID, RTC_MATRIX_COLUMN_MAJOR, &column_matrix_3x4, 0);
```

To create some motion blurred instance just pass the number of time steps and specify one matrix for each time step:

```
unsigned instID = rtcNewInstance3(sceneA, sceneB, 3);
rtcSetTransform2(sceneA, instID, RTC_MATRIX_COLUMN_MAJOR, &column_matrix_t0_3x4, 0);
rtcSetTransform2(sceneA, instID, RTC_MATRIX_COLUMN_MAJOR, &column_matrix_t1_3x4, 1);
rtcSetTransform2(sceneA, instID, RTC_MATRIX_COLUMN_MAJOR, &column_matrix_t2_3x4, 2);
```

Both scenes have to belong to the same device. One has to call `rtcCommit` on scene B before one calls `rtcCommit` on scene A. When modifying scene B one has to call `rtcUpdate` for all instances of that scene. If a ray hits the instance, then the `geomID` and `primID` members of the ray are set to the geometry ID and primitive ID of the primitive hit in scene B, and the `instID` member of the ray is set to the instance ID returned from the `rtcNewInstance3` function.

Some special care has to be taken when using user geometries and instances in the same scene. Instantiated user geometries should not set the `instID` field of the ray as this field is managed by the instancing already. However, non-instantiated user geometries should clear the `instID` field to `RTC_INVALID_GEOMETRY_ID`, to later distinguish them from instantiated geometries that have the `instID` field set.

The `rtcSetTransform2` call can be passed an affine transformation matrix with different data layouts:

Table 4.8 – Matrix layouts for `rtcSetTransform2`.

Layout	Description
RTC_MATRIX_ROW_MAJOR	The 3×4 float matrix is laid out in row major form.
RTC_MATRIX_COLUMN_MAJOR	The 3×4 float matrix is laid out in column major form.
RTC_MATRIX_COLUMN_MAJOR_ALIGNED16	The 3×4 float matrix is laid out in column major form, with each column padded by an additional 4th component.

Passing homogeneous 4×4 matrices is possible as long as the last row is (0, 0, 0, 1). If this homogeneous matrix is laid out in row major form, use the `RTC_MATRIX_ROW_MAJOR` layout. If this homogeneous matrix is laid out in column major form, use the `RTC_MATRIX_COLUMN_MAJOR_ALIGNED16` mode. In both cases, Embree will ignore the last row of the matrix.

The transformation passed to `rtcSetTransform2` transforms from the local space of the instantiated scene to world space.

See tutorial Instanced Geometry for an example of how to use instances.

Ray Layout

The ray layout to be passed to the ray tracing core is defined in the `embree2/rtcore_ray.h` header file. It is up to the user to use the ray structures defined in that file, or resemble the exact same binary data layout with their own vector classes. The ray layout might change with new Embree releases as new features get added, however, will stay constant as long as the major Embree release number does not change. The ray contains the following data members:

Member	In/Out	Description
org	in	ray origin
dir	in	ray direction (can be unnormalized)
tnear	in	start of ray segment
tfar	in/out	end of ray segment, set to hit distance after intersection
time	in	time used for multi-segment motion blur [0,1]
mask	in	ray mask to mask out geometries
Ng	out	unnormalized geometry normal in object space
u	out	barycentric u-coordinate of hit
v	out	barycentric v-coordinate of hit
geomID	out	geometry ID of hit geometry
primID	out	primitive ID of hit primitive
instID	out	instance ID of hit instance

Table 4.9 – Data fields of a ray.

This structure is in struct of array layout (SOA) for API functions accepting ray packets.

To create a single ray you can use the `RTCRay` ray type defined in `embree2/rtcore_ray.h`. To generate a ray packet of size 4, 8, or 16 you can use the `RTCRay4`, `RTCRay8`, or `RTCRay16` types. Alternatively you can also use the `RTCRayNt` template to generate ray packets of an arbitrary compile time known size.

When the ray packet size is not known at compile time (e.g. when Embree returns a ray packet in the `RTCFilterFuncN` callback function), then you can use the helper functions defined in `embree2/rtcore_ray.h` to access ray packet components:

```
float& RTCRayN_org_x(RTCRayN* rays, size_t N, size_t i);
float& RTCRayN_org_y(RTCRayN* rays, size_t N, size_t i);
float& RTCRayN_org_z(RTCRayN* rays, size_t N, size_t i);

float& RTCRayN_dir_x(RTCRayN* rays, size_t N, size_t i);
float& RTCRayN_dir_y(RTCRayN* rays, size_t N, size_t i);
float& RTCRayN_dir_z(RTCRayN* rays, size_t N, size_t i);

float& RTCRayN_tnear(RTCRayN* rays, size_t N, size_t i);
float& RTCRayN_tnear(RTCRayN* rays, size_t N, size_t i);

float&    RTCRayN_time(RTCRayN* ptr, size_t N, size_t i);
unsigned& RTCRayN_mask(RTCRayN* ptr, size_t N, size_t i);

float& RTCRayN_Ng_x(RTCRayN* ptr, size_t N, size_t i);
float& RTCRayN_Ng_y(RTCRayN* ptr, size_t N, size_t i);
float& RTCRayN_Ng_z(RTCRayN* ptr, size_t N, size_t i);

float& RTCRayN_u   (RTCRayN* ptr, size_t N, size_t i);
float& RTCRayN_v   (RTCRayN* ptr, size_t N, size_t i);

unsigned& RTCRayN_instID(RTCRayN* ptr, size_t N, size_t i);
unsigned& RTCRayN_geomID(RTCRayN* ptr, size_t N, size_t i);
unsigned& RTCRayN_primID(RTCRayN* ptr, size_t N, size_t i);
```

These helper functions get a pointer to the ray packet (`rays` parameter), the

packet size N, and returns a reference to some component (e.g. x-component of origin) of the the ith ray of the packet.

Please note that there is some incompatibility in the layout of a single ray (RTCRay type) and a ray packet of size 1 (RTCRayNt<1> type) as the org and dir component are aligned to 16 bytes for single rays (see embree2/rtcore_ray.h). This incompatibility will get resolved in a future release, but has to be maintained for compatibility currently. Until then, the ray stream API will always use the single ray layout RTCRay for rays packets of size N=1, and the RTCRayNt layout for ray packets of size not equal 1. The helper functions above to access a ray packet of size N take care of this incompatibility.

Some callback functions get passed a hit structure with the following data members:

Member	In/Out	Description
instID	in	instance ID of hit instance
geomID	in	geometry ID of hit geometry
primID	in	primitive ID of hit primitive
u	in	barycentric u-coordinate of hit
v	in	barycentric v-coordinate of hit
t	in	hit distance
Ng	in	unnormalized geometry normal in object space

Table 4.10 – Data fields of a hit.

This structure is in struct of array layout (SOA) for hit packets of size N. The layout of a hit packet of size N is defined by the RTCHitNt template in embree2/rtcore_ray.h.

When the hit packet size is not known at compile time (e.g. when Embree returns a hit packet in the RTCFilterFuncN callback function), you can use the helper functions defined in embree2/rtcore_ray.h to access hit packet components:

```
unsigned& RTCHitN_instID(RTCHitN* hits, size_t N, size_t i);
unsigned& RTCHitN_geomID(RTCHitN* hits, size_t N, size_t i);
unsigned& RTCHitN_primID(RTCHitN* hits, size_t N, size_t i);

float& RTCHitN_u   (RTCHitN* hits, size_t N, size_t i);
float& RTCHitN_v   (RTCHitN* hits, size_t N, size_t i);
float& RTCHitN_t   (RTCHitN* hits, size_t N, size_t i);

float& RTCHitN_Ng_x(RTCHitN* hits, size_t N, size_t i);
float& RTCHitN_Ng_y(RTCHitN* hits, size_t N, size_t i);
float& RTCHitN_Ng_z(RTCHitN* hits, size_t N, size_t i);
```

These helper functions get a pointer to the hit packet (hits parameter), the packet size N, and returns a reference to some component (e.g. u-component) of the the ith hit of the packet.

Ray Queries

The API supports finding the closest hit of a ray segment with the scene (rtcIntersect functions), and determining if any hit between a ray segment and the scene exists (rtcOccluded functions).

Normal Mode

In normal mode the following API functions should be used to trace rays:

```
void rtcIntersect  (                   RTCScene scene, RTCRay&    ray);
void rtcIntersect4 (const void* valid, RTCScene scene, RTCRay4&   ray);
void rtcIntersect8 (const void* valid, RTCScene scene, RTCRay8&   ray);
void rtcIntersect16(const void* valid, RTCScene scene, RTCRay16&  ray);
void rtcOccluded   (                   RTCScene scene, RTCRay&    ray);
void rtcOccluded4  (const void* valid, RTCScene scene, RTCRay4&   ray);
void rtcOccluded8  (const void* valid, RTCScene scene, RTCRay8&   ray);
void rtcOccluded16 (const void* valid, RTCScene scene, RTCRay16&  ray);
```

The `rtcIntersect` and `rtcOccluded` function operate on single rays. The `rtcIntersect4` and `rtcOccluded4` functions operate on ray packets of size 4. The `rtcIntersect8` and `rtcOccluded8` functions operate on ray packets of size 8, and the `rtcIntersect16` and `rtcOccluded16` functions operate on ray packets of size 16.

For the ray packet mode with packet size of 4, 8, or 16, the user has to provide a pointer to 4, 8, or 16 of 32 bit integers that act as a ray activity mask (`valid` argument). If one of these integers is set to `0x00000000` the corresponding ray is considered inactive and if the integer is set to `0xFFFFFFFF`, the ray is considered active. Rays that are inactive will not update any hit information.

Finding the closest hit distance is done through the `rtcIntersect` type functions. These get the activity mask (`valid` parameter), the scene (`scene` parameter), and a ray as input (`ray` parameter). The layout of the ray structure is described in Section Ray Layout. The user has to initialize the ray origin (`org`), ray direction (`dir`), and ray segment (`tnear`, `tfar`). The ray segment has to be in the range $[0, \infty]$, thus ranges that start behind the ray origin are not valid, but ranges can reach to infinity. The implementation makes no guarantees if primitives whose hit distance is exactly at (or very close to) `tnear` or `tfar` are hit or missed. If you want to exclude intersections at `tnear` just pass a slighly enlarged `tnear` and if you want to include intersections at `tfar` pass a slighly enlarged `tfar` to Embree. The geometry ID (`geomID` member) has to get initialized to `RTC_INVALID_GEOMETRY_ID` (-1). If the scene contains instances, also the instance ID (`instID`) has to get initialized to `RTC_INVALID_GEOMETRY_ID` (-1). If the scene contains motion blur geometries, also the ray time (`time`) has to get initialized to a value in the range $[0, 1]$. If ray masks are enabled at compile time, also the ray mask (`mask`) has to get initialized. After tracing the ray, the hit distance (`tfar`), geometry normal (`Ng`)[1], local hit coordinates (`u`, `v`), geometry ID (`geomID`), and primitive ID (`primID`) are set. If the scene contains instances, also the instance ID (`instID`) is set, if an instance is hit. The geometry ID corresponds to the ID returned at creation time of the hit geometry, and the primitive ID corresponds to the nth primitive of that geometry, e.g. nth triangle. The instance ID corresponds to the ID returned at creation time of the instance.

[1] Note that `Ng` is in object space and needs to be transformed to world space for instanced scenes.

Testing if any geometry intersects with the ray segment is done through the `rtcOccluded` functions. Initialization has to be done as for `rtcIntersect`. If some geometry got found along the ray segment, the geometry ID (`geomID`) will get set to 0. Other hit information of the ray is undefined after calling `rtcOccluded`.

In normal mode, data alignment requirements for ray query functions operating on single rays is 16 bytes for the ray. Data alignment requirements for query functions operating on AOS packets of 4, 8, or 16 rays, is 16, 32, and 64 bytes respectively, for the valid mask and the ray. To operate on packets of 4 rays, the CPU has to support SSE, to operate on packets of 8 rays, the CPU has to support AVX, and to operate on packets of 16 rays, the CPU has to support AVX-512 instructions. Additionally, the required ISA has to be enabled in Embree at compile time to use the desired packet size.

The following code shows an example of setting up a single ray and traces it through the scene:

```
RTCRay ray;
ray.org = ray_origin;
ray.dir = ray_direction;
ray.tnear = 0.0f;
ray.tfar = inf;
ray.instID = RTC_INVALID_GEOMETRY_ID;
ray.geomID = RTC_INVALID_GEOMETRY_ID;
ray.primID = RTC_INVALID_GEOMETRY_ID;
ray.mask = 0xFFFFFFFF;
ray.time = 0.0f;
rtcIntersect(scene, ray);
```

See tutorial Triangle Geometry for a complete example of how to trace rays.

Ray Stream Mode

For the stream mode new functions got introduced that operate on streams of rays:

```
void rtcIntersect1M    (RTCScene scene, const RTCIntersectContext* context,
                        RTCRay* rays, size_t M, size_t stride);
void rtcIntersect1Mp   (RTCScene scene, const RTCIntersectContext* context,
                        RTCRay**rays, size_t M);
void rtcIntersectNM    (RTCScene scene, const RTCIntersectContext* context,
                        RTCRayN* rays, size_t N, size_t M, size_t stride);
void rtcIntersectNp    (RTCScene scene, const RTCIntersectContext* context,
                        RTCRayNp& rays, size_t N);

void rtcOccluded1M     (RTCScene scene, const RTCIntersectContext* context,
                        RTCRay* rays, size_t M, size_t stride);
void rtcOccluded1Mp    (RTCScene scene, const RTCIntersectContext* context,
                        RTCRay** rays, size_t M);
void rtcOccludedNM     (RTCScene scene, const RTCIntersectContext* context,
                        RTCRayN* rays, size_t N, size_t M, size_t stride);
void rtcOccludedNp     (RTCScene scene, const RTCIntersectContext* context,
                        RTCRayNp& rays, size_t N, size_t flags);
```

The `rtcIntersectNM` and `rtcOccludedNM` ray stream functions operate on an array of `M` ray packets of packet size `N`. The offset in bytes between consecutive ray packets can be specified by the `stride` parameter. Data alignment requirements for ray streams is 16 bytes. The packet size `N` has to be larger than 0 and the stream size `M` can be an arbitrary positive integer including 0. Tracing for example a ray stream consisting of four 8-wide SOA ray packets just requires to set the parameters `N` to 8, `M` to 4 and the `stride` to `sizeof(RTCRay8)`. A ray in a ray stream is considered inactive during traversal/intersection if its `tnear` value is larger than its `tfar` value.

The ray streams functions `rtcIntersect1M` and `rtcOccluded1M` are just a shortcut for single ray streams with a packet size of `N=1`. `rtcIntersect1Mp` and `rtcOccluded1Mp` are similar to `rtcIntersect1M` and `rtcOccluded1M` while taking a stream of pointers to single rays as input. The `rtcIntersectNp` and `rtcOccludedNp` functions do not require the individual components of the SOA ray packets to be stored sequentially in memory, but at different adresses as specified in the `RTCRayNp` structure.

The intersection context passed to the stream version of the ray query functions, can specify some intersection flags to optimize traversal and a `userRayExt`

pointer that can be used to extent the ray with additional data as described in Section Extending the Ray Structure. The intersection context is propagated to each stream user callback function invoked.

```
struct RTCIntersectContext
{
  RTCIntersectFlags flags;   //!< intersection flags
  void* userRayExt;          //!< can be used to pass extended ray data to callbacks
};
```

As intersection flag the user can currently specify if Embree should optimize traversal for coherent or incoherent ray distributions.

```
enum RTCIntersectFlags
{
  RTC_INTERSECT_COHERENT   = 0,  //!< optimize for coherent rays
  RTC_INTERSECT_INCOHERENT = 1   //!< optimize for incoherent rays
};
```

The following code shows an example of setting up a stream of single rays and tracing it through the scene:

```
RTCRay rays[128];

/* first setup all rays */
for (size_t i=0; i<128; i++)
{
  rays[i].org = calculate_ray_org(i);
  rays[i].dir = calculate_ray_dir(i);
  rays[i].tnear = 0.0f;
  rays[i].tfar = inf;
  rays[i].instID = RTC_INVALID_GEOMETRY_ID;
  rays[i].geomID = RTC_INVALID_GEOMETRY_ID;
  rays[i].primID = RTC_INVALID_GEOMETRY_ID;
  rays[i].mask = 0xFFFFFFFF;
  rays[i].time = 0.0f;
}

/* now create a context and trace the ray stream */
RTCIntersectContext context;
context.flags = RTC_INTERSECT_INCOHERENT;
context.userRayExt = nullptr;
rtcIntersectNM(scene, &context, &rays, 1, 128, sizeof(RTCRay));
```

See tutorial Stream Viewer for a complete example of how to trace ray streams.

Interpolation of Vertex Data

Smooth interpolation of per-vertex data is supported for triangle meshes, quad meshs, hair geometry, line segment geometry, and subdivision geometry using the `rtcInterpolate2` API call. This interpolation function does ignore displacements and always interpolates the underlying base surface.

```
void rtcInterpolate2(RTCScene scene,
                     unsigned geomID, unsigned primID,
                     float u, float v,
```

```
                      RTCBufferType buffer,
                      float* P,
                      float* dPdu, float* dPdv,
                      float* ddPdudu, float* ddPdvdv, float* ddPdudv,
                      size_t numFloats);
```

This call smoothly interpolates the per-vertex data stored in the specified geometry buffer (`buffer` parameter) to the u/v location (`u` and `v` parameters) of the primitive (`primID` parameter) of the geometry (`geomID` parameter) of the specified scene (`scene` parameter). The interpolation buffer (`buffer` parameter) has to contain (at least) `numFloats` floating point values per vertex to interpolate. As interpolation buffer one can specify the `RTC_VERTEX_BUFFER0` and `RTC_VERTEX_BUFFER1` as well as one of two special user vertex buffers `RTC_USER_VERTEX_BUFFER0` and `RTC_USER_VERTEX_BUFFER1`. These user vertex buffers can only get set using the `rtcSetBuffer2` call, they cannot get managed internally by Embree as they have no default layout. The last element of the buffer has to be padded to 16 bytes, such that it can be read safely using SSE instructions.

The `rtcInterpolate` call stores `numFloats` interpolated floating point values to the memory location pointed to by `P`. One can avoid storing the interpolated value by setting `P` to NULL.

The first order derivative of the interpolation by u and v are stored at the `dPdu` and `dPdv` memory locations. One can avoid storing first order derivatives by setting both `dPdu` and `dPdv` to NULL.

The second order derivatives are stored at the `ddPdudu`, `ddPdvdv`, and `ddPdudv` memory locations. One can avoid storing second order derivatives by setting these three pointers to NULL.

The `RTC_INTERPOLATE` algorithm flag of a scene has to be enabled to perform interpolations.

It is explicitly allowed to call this function on disabled geometries. This makes it possible to use a separate subdivision mesh with different vertex creases, edge creases, and boundary handling for interpolation of texture coordinates if that is necessary.

The applied interpolation will do linear interpolation for triangle and quad meshes, linear interpolation for line segments, cubic Bézier interpolation for hair, and apply the full subdivision rules for subdivision geometry.

There is also a second interpolate call `rtcInterpolateN2` that can be used for ray packets.

```
void rtcInterpolateN2(RTCScene scene, unsigned geomID,
                      const void* valid, const unsigned* primIDs,
                      const float* u, const float* v, size_t numUVs,
                      RTCBufferType buffer,
                      float* dP,
                      float* dPdu, float* dPdv,
                      float* ddPdudu, float* ddPdvdv, float* ddPdudv,
                      size_t numFloats);
```

This call is similar to the first version, but gets passed `numUVs` many u/v coordinates and a valid mask (`valid` parameter) that specifies which of these coordinates are valid. The valid mask points to `numUVs` integers and a value of -1 denotes valid and 0 invalid. If the valid pointer is NULL all elements are considers valid. The destination arrays are filled in structure of array (SoA) layout.

See tutorial Interpolation for an example of using the `rtcInterpolate2` function.

Buffer Sharing

Embree supports sharing of buffers with the application. Each buffer that can be mapped for a specific geometry can also be shared with the application, by passing a pointer, offset, stride, and number of elements of the application side buffer using the `rtcSetBuffer2` API function.

```
void rtcSetBuffer2(RTCScene scene, unsigned geomID, RTCBufferType type,
                   void* ptr, size_t offset, size_t stride, size_t size);
```

The `rtcSetBuffer2` function has to get called before any call to `rtcMapBuffer` for that buffer, otherwise the buffer will get allocated internally and the call to `rtcSetBuffer2` will fail. The buffer has to remain valid as long as the geometry exists, and the user is responsible to free the buffer when the geometry gets deleted. When a buffer is shared, it is safe to modify that buffer without mapping and unmapping it. However, for dynamic scenes one still has to call `rtcUpdate` for modified geometries and the buffer data has to stay constant from the `rtcCommit` call to after the last ray query invocation.

The `offset` parameter specifies a byte offset to the start of the first element, the `stride` parameter specifies a byte stride between the different elements of the shared buffer and the `size` parameter specified the number of elements stored inside the buffer. This support for offset and stride allows the application quite some freedom in the data layout of these buffers, however, some restrictions apply. Index buffers always store 32 bit indices and vertex buffers always store single precision floating point data. The start address `ptr+offset` and `stride` always have to be aligned to 4 bytes, otherwise the `rtcSetBuffer2` function will fail. The `size` parameter can be used to change the size of a buffer, which makes it possible to change the number of elements inside a mesh (by changing the size of the `RTC_INDEX_BUFFER`).

For vertex buffers (`RTC_VERTEX_BUFFER` and `RTC_USER_VERTEX_BUFFER`), the last element must be readable using SSE instructions, thus padding the last element to 16 bytes size is required for some layouts.

The following is an example of how to create a mesh with shared index and vertex buffers:

```
unsigned geomID = rtcNewTriangleMesh(scene, geomFlags, numTriangles, numVertices);
rtcSetBuffer2(scene, geomID, RTC_VERTEX_BUFFER, vertexPtr, 0, 3*sizeof(float), numVertices);
rtcSetBuffer2(scene, geomID, RTC_INDEX_BUFFER, indexPtr, 0, 3*sizeof(int), numTriangles);
```

Sharing buffers can significantly reduce the memory required by the application, thus we recommend using this feature. When enabling the `RTC_COMPACT` scene flag, the spatial index structures of Embree might also share the vertex buffer, resulting in even higher memory savings.

Multi-Segment Motion Blur

All geometry types support multi-segment motion blur with equidistant time steps and arbitrary number of time steps in the range of 2 to 129. Each geometry can have a different number of time steps. Some motion blur geometry is constructed by passing the number of time steps to the geometry construction function and setting the vertex arrays `RTC_VERTEX_BUFFER0+t` for each time step `t`:

```
unsigned geomID = rtcNewTriangleMesh(scene, geomFlags, numTris, numVertices, 3);
rtcSetBuffer2(scene, geomID, RTC_VERTEX_BUFFER0+0, vertex0Ptr, 0, sizeof(Vertex), numVertices);
rtcSetBuffer2(scene, geomID, RTC_VERTEX_BUFFER0+1, vertex1Ptr, 0, sizeof(Vertex), numVertices);
rtcSetBuffer2(scene, geomID, RTC_VERTEX_BUFFER0+2, vertex2Ptr, 0, sizeof(Vertex), numVertices);
rtcSetBuffer2(scene, geomID, RTC_INDEX_BUFFER, indexPtr, 0, sizeof(Triangle), numTris);
```

If a scene contains geometries with motion blur, the user has to set the `time` member of the ray to a value in the range $[0, 1]$. The motion blur geometry is defined by linearly interpolating the geometries of neighboring time steps. Each ray can specify a different time, even inside a ray packet.

User Data Pointer

A user data pointer can be specified and queried per geometry, to efficiently map from the geometry ID returned by ray queries to the application representation for that geometry.

```
void  rtcSetUserData (RTCScene scene, unsigned geomID, void* ptr);
void* rtcGetUserData (RTCScene scene, unsigned geomID);
```

The user data pointer of some user defined geometry get additionally passed to the intersect and occluded callback functions of that user geometry. Further, the user data pointer is also passed to intersection filter callback functions attached to some geometry.

The `rtcGetUserData` function is on purpose not thread safe with respect to other API calls that modify the scene. Consequently, this function can be used to efficiently query the user data pointer during rendering (also by multiple threads), but should not get called while modifying the scene with other threads.

Geometry Mask

A 32 bit geometry mask can be assigned to triangle meshes and hair geometries using the `rtcSetMask` call.

```
rtcSetMask(scene, geomID, mask);
```

Only if the bitwise and operation of this mask with the mask stored inside the ray is not 0, primitives of this geometry are hit by a ray. This feature can be used to disable selected triangle mesh or hair geometries for specifically tagged rays, e.g. to disable shadow casting for some geometry. This API feature is disabled in Embree by default at compile time, and can be enabled in CMake through the `EMBREE_RAY_MASK` parameter.

Filter Functions

The API supports per geometry filter callback functions that are invoked for each intersection found during the `rtcIntersect` or `rtcOccluded` calls. The former ones are called intersection filter functions, the latter ones occlusion filter functions. The filter functions can be used to implement various useful features, such as accumulating opacity for transparent shadows, counting the number of surfaces along a ray, collecting all hits along a ray, etc. Filter functions can also be used to selectively reject hits to enable backface culling for some geometries. If the backfaces should be culled in general for all geometries then it is faster to enable `EMBREE_BACKFACE_CULLING` during compilation of Embree instead of using filter functions.

If the `RTC_SCENE_HIGH_QUALITY` mode is set, the intersection and occlusion filter functions may be called multiple times for the same hit. For some usage scenarios, the application may have to work around this by collecting already reported hits (geomID/primID pairs) and ignoring duplicates for some usage scenarios.

Normal Mode

In normal mode the filter functions provided by the user need to have the following signature:

```
void RTCFilterFunc  (                  void* userDataPtr, RTCRay&   ray);
void RTCFilterFunc4 (const void* valid, void* userDataPtr, RTCRay4&  ray);
void RTCFilterFunc8 (const void* valid, void* userDataPtr, RTCRay8&  ray);
void RTCFilterFunc16(const void* valid, void* userDataPtr, RTCRay16& ray);
```

The `valid` pointer points to an integer valid mask (0 means invalid and -1 means valid). The `userDataPtr` is a user pointer optionally set per geometry through the `rtcSetUserData` function. All hit information inside the ray is valid. If the hit geometry is instanced, the `instID` member of the ray is valid and the ray origin, direction, and geometry normal visible through the ray are in object space.

The filter function can reject a hit by setting the `geomID` member of the ray to `RTC_INVALID_GEOMETRY_ID`, otherwise the hit is accepted. The filter function is not allowed to modify the ray input data (`org`, `dir`, `time`, `mask`, and `tnear` members), but can modify the hit data of the ray (`u`, `v`, `Ng`, `tfar`, `geomID`, `primID`, and `instID` members). Updating the `tfar` distance to a smaller value is possible without limitation. However, increasing the `tfar` distance of the ray to a larger value `tfar'` , does not guarantee intersections between `tfar` and `tfar'` to be reported later, as the corresponding subtrees might have gotten culled already.

The intersection and occlusion filter functions for different ray types are set for some geometry of a scene using the following API functions:

```
void rtcSetIntersectionFilterFunction  (RTCScene, unsigned geomID, RTCFilterFunc   filter);
void rtcSetIntersectionFilterFunction4 (RTCScene, unsigned geomID, RTCFilterFunc4  filter);
void rtcSetIntersectionFilterFunction8 (RTCScene, unsigned geomID, RTCFilterFunc8  filter);
void rtcSetIntersectionFilterFunction16(RTCScene, unsigned geomID, RTCFilterFunc16 filter);

void rtcSetOcclusionFilterFunction  (RTCScene, unsigned geomID, RTCFilterFunc   filter);
void rtcSetOcclusionFilterFunction4 (RTCScene, unsigned geomID, RTCFilterFunc4  filter);
void rtcSetOcclusionFilterFunction8 (RTCScene, unsigned geomID, RTCFilterFunc8  filter);
void rtcSetOcclusionFilterFunction16(RTCScene, unsigned geomID, RTCFilterFunc16 filter);
```

The intersection and occlusion filter functions of type `RTCFilterFunc` are only called by the `rtcIntersect` and `rtcOccluded` functions. Similar the filter functions of type `FilterFunc4`, `FilterFunc8`, and `FilterFunc16` are called by `rtcIntersect4/8/16` and `rtcOccluded4/8/16` of matching width.

Stream Mode

For ray stream mode a new type of filter function `RTCFilterFuncN` got introduced:

```
void RTCFilterFuncN (int* valid,
                     void* userDataPtr,
                     const RTCIntersectContext* context,
                     RTCRayN* ray,
                     const RTCHitN* potentialHit,
                     const size_t N);
```

The stream intersection and occlusion filter functions of this new type are set for some geometry of a scene using the following API functions:

```
void rtcSetIntersectionFilterFunctionN (RTCScene, unsigned geomID, RTCFilterFuncN filter);
void rtcSetOcclusionFilterFunctionN    (RTCScene, unsigned geomID, RTCFilterFuncN filter);
```

For the callback RTCFilterFuncN, the valid parameter points to an integer valid mask (0 means invalid and -1 means valid). The userDataPtr is a user pointer optionally set per geometry through the rtcSetUserData function. The context parameter points to the intersection context passed to the ray query function. The ray parameter contains the current ray. All hit data inside the ray are undefined, except the tfar value. The potentialHit parameter points to the new hit to test and update. The N parameter is the number of rays and hits found in the ray and potentialHit. If the hit geometry is instanced, the instID member of the ray is valid and the ray as well as the potential hit are in object space.

As the ray packet size N can be arbitrary, the ray and hit should get accessed through the helper functions as describe in Section Ray Layout.

The callback function has the task to check for each valid ray whether it wants to accept or reject the corresponding hit. To reject a hit, the filter callback function just has to write 0 to the integer valid mask of the corresponding ray. The filter function is not allowed to modify the ray input data (org, dir, time, mask, and tnear members), nor the potential hit, nor inactive components.

An intersection filter callback function can accept a hit by updating all hit data members of the ray (u, v, Ng, tfar, geomID, primID, and instID members) and keep the valid mask set to -1.

An occlusion filter callback function can accept a hit by setting the geomID member of the ray to 0 and keep the valid mask set to -1.

The intersection filter callback of most applications will just copy the potentialHit into the appropiate fields of the ray, but this is not a requirement and the hit data of the ray can get modified arbitrarily. Updating the tfar distance to a smaller value (e.g. the t distance of the potential hit) is possible without limitation. However, increasing the tfar distance of the ray to a larger value tfar' , does not guarantee intersections between tfar and tfar' to be reported later, as the corresponding subtrees might have gotten culled already.

Displacement Mapping Functions

The API supports displacement mapping for subdivision meshes. A displacement function can be set for some subdivision mesh using the rtcSetDisplacementFunction API call.

```
void rtcSetDisplacementFunction2(RTCScene, unsigned geomID, RTCDisplacementFunc, RTCBounds*);
```

A displacement function of NULL will delete an already set displacement function. The bounds parameter is optional. If NULL is passed as bounds, then the displacement shader will get evaluated during the build process to properly bound displaced geometry. If a pointer to some bounds of the displacement are passed, then the implementation can choose to use these bounds to bound displaced geometry. When bounds are specified, then these bounds have to be conservative and should be tight for best performance.

The displacement function has to have the following type:

```
typedef void (*RTCDisplacementFunc2)(void* ptr,
                                      unsigned geomID, unsigned primID, unsigned timeStep,
                                      const float* u,  const float* v,
                                      const float* nx, const float* ny, const float* nz,
                                      float* px, float* py, float* pz,
                                      size_t N);
```

The displacement function is called with the user data pointer of the geometry (ptr), the geometry ID (geomID), and primitive ID (primID) of a patch to

displace. For motion blur the time step `timeStep` is also specified, such that the function can be time varying. For the patch, a number N of points to displace are specified in a struct of array layout. For each point to displace the local patch UV coordinates (`u` and `v` arrays), the normalized geometry normal (`nx`, `ny`, and `nz` arrays), as well as world space position (`px`, `py`, and `pz` arrays) are provided. The task of the displacement function is to use this information and move the world space position inside the allowed specified bounds around the point.

All passed arrays are guaranteed to be 64 bytes aligned, and properly padded to make wide vector processing inside the displacement function possible.

The displacement mapping functions might get called during the `rtcCommit` call, or lazily during the `rtcIntersect` or `rtcOccluded` calls.

Also see tutorial Displacement Geometry for an example of how to use the displacement mapping functions.

Extending the Ray Structure

Normal Mode

If Embree is used in normal mode, the ray passed to the filter callback functions and user geometry callback functions is guaranteed to be the same ray pointer initially provided to the ray query function by the user. For that reason, it is safe to extend the ray by additional data and access this data inside the filter callback functions (e.g. to accumulate opacity) and user geometry callback functions.

Stream Mode

If Embree is used in stream mode, the ray passed to the filter callback and user geometry callback functions is `not` guaranteed to be the same ray pointer initially passed to the ray query function, as the stream implementation may decide to copy rays around, reorder them, and change the data layout internally when appropiate (e.g. SOA to AOS conversion).

To identify specific rays in the callback functions, the user has to pass an ID with each ray and set the `userRayExt` member of the intersection context to point to its ray extensions. The ray extensions can be stored in a seprarate memory location but also just after the end of each ordinary ray (or ray packet). In the latter case, you can just point the `userRayExt` to the input rays.

To encode a ray ID the ray mask field can be used entirely when the ray mask feature is disabled, or unused bits of the ray mask can be used in case the ray mask feature is enabled (e.g. by using the lower 16 bits as ray ID, and the upper 16 bits as ray mask, and setting the lower 16 bits of each geometry mask always to 0).

The intersection context provided to the stream ray query functions is passed to each stream callback function (e.g. `RTCIntersectFuncN`, `RTCIntersectFunc1Mp`, or `RTCFilterFuncN`). Thus, in the callback function, the ray ID can get decoded, and the extended ray data accessed through the `userRayExt` pointer stored inside the intersection context. For SPMD type programs this access requires `gather` and `scatter` operations to access the user ray extensions.

Not that using the ray ID to access the ray extensions is necessary, as the ray IDs might have changed from the IDs passed to the ray query function. E.g. if you trace a ray packet with 8 rays 0 to 8, then even if a callback gets called with a ray packet of 8 rays, they rays might have gotten reordered. Further, the callback might get called with a subpacket of a size smaller than 8 (e.g. `N=5`). However, optimizing for the common case in which Embree keeps such a packet intact (thus having a special codepath for `N=8` and unchanged IDs) can give higher performance.

Sharing Threads with Embree

On some implementations, Embree supports using the application threads when building internal data structures, by using the

```
void rtcCommitThread(RTCScene, unsigned threadIndex, unsigned threadCount);
```

API call to commit the scene. This function has to get called by all threads that want to cooperate in the scene commit. Each call is provided the scene to commit, the index of the calling thread in the range [0, `threadCount`-1], and the number of threads that will call into this commit operation for the scene. All threads will return again from this function after the scene commit is finished.

Multiple such scene commit operations can also be running at the same time, e.g. it is possible to commit many small scenes in parallel using one thread per commit operation. Subsequent commit operations for the same scene can use different number of threads in the `rtcCommitThread` or use the Embree internal threads using the `rtcCommit` call.

Note: When using Embree with the Intel® Threading Building Blocks (which is the default) you should not use the `rtcCommitThread` function. Sharing of your threads with TBB is not possible and TBB will always generate its own set of threads. We recommend to also use TBB inside your application to share threads with the Embree library. When using TBB inside your application do never use the `rtcCommitThread` function.

Note: When enabling the Embree internal tasking system the `rtcCommitThread` feature will work as expected and use the application threads for hierarchy building.

Join Build Operation

The special `rtcCommitJoin` function can be used from multiple threads to join a scene build operation. All thread have to consistently call `rtcCommitJoin` and no other commit variant.

This feature allows a flexible way to lazily create hierarchies during rendering. A thread reaching a not yet constructed sub-scene of a two-level scene, can generate the sub-scene geometry and call `rtcCommitJoin` on that just generated scene. During construction, further threads reaching the not-yet-built scene, can join the build operation by also invoking `rtcCommitJoin`. A thread that calls `rtcCommitJoin` after the build finishes, will directly return from the `rtcCommitJoin` call (even for static scenes).

Note: When using Embree with the Intel® Threading Building Blocks, thread that call `rtcCommitJoin` will join the build operation, but other TBB worker threads might also participate in the build. To avoid thread oversubscription, we recommend using TBB also inside the application. Further, the join mode only works properly starting with TBB v4.4 Update 1. For earlier TBB versions threads that call `rtcCommitJoin` to join a running build will just wait for the build to finish.

Note: When using Embree with the internal tasking system, exclusively threads that call `rtcCommitJoin` will perform the build operation, and no additional worker threads are scheduled.

Memory Monitor Callback

Using the memory monitor callback mechanism, the application can track the memory consumption of an Embree device, and optionally terminate API calls that consume too much memory.

The user provided memory monitor callback function must have the following signature:

```
bool (*RTCMemoryMonitorFunc2)(void* userPtr, const ssize_t bytes, const bool post);
```

A single such callback function per device can be registered by calling

```
rtcDeviceSetMemoryMonitorFunction2(RTCDevice device, RTCMemoryMonitorFunc2 func, void* userPtr);
```

and deregistered again by calling it with `NULL` as function pointer. Once registered the Embree device will invoke the callback function before or after it allocates or frees important memory blocks. The `userPtr` value that is set at registration time is passed to each invokation of the callback function. The callback function might get called from multiple threads concurrently.

The application can track the current memory usage of the Embree device by atomically accumulating the provided `bytes` input parameter. This parameter will be >0 for allocations and <0 for deallocations. The `post` input parameter is true if the callback function was invoked after the allocation or deallocation, otherwise it is false.

Embree will continue its operation normally when returning true from the callback function. If false is returned, Embree will cancel the current operation with the RTC_OUT_OF_MEMORY error code. Cancelling will only happen when the callback was called for allocations (bytes > 0), otherwise the cancel request will be ignored. If a callback that was invoked before the allocation happens (`post == false`) cancels the operation, then the `bytes` parameter should not get accumulated, as the allocation will never happen. If a callback that was called after the allocation happened (`post == true`) cancels the operation, then the `bytes` parameter should get accumulated, as the allocation properly happened. Issuing multiple cancel requests for the same operation is allowed.

Progress Monitor Callback

The progress monitor callback mechanism can be used to report progress of hierarchy build operations and to cancel long lasting build operations.

The user provided progress monitor callback function has to have the following signature:

```
bool (*RTCProgressMonitorFunc)(void* userPtr, const double n);
```

A single such callback function can be registered per scene by calling

```
rtcSetProgressMonitorFunction(RTCScene, RTCProgressMonitorFunc, void* userPtr);
```

and deregistered again by calling it with `NULL` for the callback function. Once registered Embree will invoke the callback function multiple times during hierarchy build operations of the scene, by providing the `userPtr` pointer that was set at registration time, and a double `n` in the range $[0, 1]$ estimating the completion amount of the operation. The callback function might get called from multiple threads concurrently.

When returning `true` from the callback function, Embree will continue the build operation normally. When returning `false` Embree will cancel the build operation with the RTC_CANCELLED error code. Issuing multiple cancel requests for the same build operation is allowed.

Configuring Embree

Some internal device parameters can be set and queried using the `rtcDeviceSetParameter1i` and `rtcDeviceGetParameter1i` API call. The parameters from the following table are available to set/query:

Table 4.11 – Parameters for `rtcDeviceSetParameter` and `rtcDeviceGetParameter`.

Parameter	Description	Read/Write
RTC_CONFIG_VERSION_MAJOR	returns Embree major version	Read only
RTC_CONFIG_VERSION_MINOR	returns Embree minor version	Read only
RTC_CONFIG_VERSION_PATCH	returns Embree patch version	Read only
RTC_CONFIG_VERSION	returns Embree version as integer e.g. Embree v2.8.2 → 20802	Read only
RTC_CONFIG_INTERSECT1	checks if `rtcIntersect1` is supported	Read only
RTC_CONFIG_INTERSECT4	checks if `rtcIntersect4` is supported	Read only
RTC_CONFIG_INTERSECT8	checks if `rtcIntersect8` is supported	Read only
RTC_CONFIG_INTERSECT16	checks if `rtcIntersect16` is supported	Read only
RTC_CONFIG_INTERSECT_STREAM	checks if `rtcIntersect1M`, `rtcIntersect1Mp`, `rtcIntersectNM`, and `rtcIntersectNp` are supported	Read only
RTC_CONFIG_TRIANGLE_GEOMETRY	checks if triangle geometries are supported	Read only
RTC_CONFIG_QUAD_GEOMETRY	checks if quad geometries are supported	Read only
RTC_CONFIG_LINE_GEOMETRY	checks if line geometries are supported	Read only
RTC_CONFIG_HAIR_GEOMETRY	checks if hair geometries are supported	Read only
RTC_CONFIG_SUBDIV_GEOMETRY	checks if subdivision meshes are supported	Read only
RTC_CONFIG_USER_GEOMETRY	checks if user geometries are supported	Read only
RTC_CONFIG_RAY_MASK	checks if ray masks are supported	Read only
RTC_CONFIG_BACKFACE_CULLING	checks if backface culling is supported	Read only
RTC_CONFIG_INTERSECTION_FILTER	checks if intersection filters are enabled	Read only
RTC_CONFIG_INTERSECTION_FILTER_RESTORE	checks if intersection filters restore previous hit	Read only
RTC_CONFIG_IGNORE_INVALID_RAYS	checks if invalid rays are ignored	Read only
RTC_CONFIG_TASKING_SYSTEM	return used tasking system (0 = INTERNAL, 1 = TBB)	Read only
RTC_SOFTWARE_CACHE_SIZE	Configures the software cache size (used to cache subdivision surfaces for instance). The size is specified as an integer number of bytes. The software cache cannot be configured during rendering.	Write only
RTC_CONFIG_COMMIT_JOIN	Checks if rtcCommit can be used to join build operation (not supported when Embree is compiled with some older TBB versions)	Read only
RTC_CONFIG_COMMIT_THREAD	Checks if rtcCommitThread is available (not supported when Embree is compiled with some older TBB versions)	Read only

For example, to configure the size of the internal software cache that is used to handle subdivision surfaces use the `RTC_SOFTWARE_CACHE_SIZE` parameter to set desired size of the cache in bytes:

```
rtcDeviceSetParameter1i(device, RTC_SOFTWARE_CACHE_SIZE, bytes);
```

The software cache cannot get configured while any Embree API call is executed. Best configure the size of the cache only once at application start.

Limiting number of Build Threads

You can use the TBB API to limit the number of threads used by Embree during hierarchy construction. Therefore just create a global taskscheduler_init object, initialized with the number of threads to use:

```
#include <tbb/tbb.h>

tbb::task_scheduler_init init(numThreads);
```

Thread Creation and Affinity Settings

Tasking systems like TBB create worker threads on demand which will add a runtime overhead for the very first `rtcCommit` call. In case you want to benchmark the scene build time, you should start threads at application startup. You can let Embree start TBB threads by passing `start_threads=1` to the init parameter of `rtcNewDevice`.

On machines with a high thread count (e.g. dual-socket Xeon or Xeon Phi machines), affinitizing TBB worker threads increases build and rendering performance. You can let Embree affinitize TBB worker threads by passing `set_affinity=1` to the init parameter of `rtcNewDevice`.

All Embree tutorials automatically start and affinitize TBB worker threads by passing `start_threads=1,set_affinity=1` to `rtcNewDevice`.

Huge Page Support

Embree supports 2MB huge pages under Windows, Linux, and MacOSX. Under Linux huge page support is enabled by default and under Windows and MacOSX disabled by default. Huge page support can get enabled in Embree by passing `hugepages=1` to `rtcNewDevice` or disabled by passing `hugepages=0` to `rtcNewDevice`.

We recommend using 2MB huge pages with Embree under Linux as this improves ray tracing performance by about 5 - 10%. Under Windows using huge pages requires the application to run in elavated mode which is a security issue. Under MacOSX huge pages are rarely available as memory tends to get quickly fragmented.

Huge Pages under Windows

To use huge pages under Windows, the current user must have the “Lock pages in memory” (SeLockMemoryPrivilege) assigned. This can be configured through the “Local Security Policy” application, by adding a user to “Local Policies” -> “User Rights Assignment” -> “Lock pages in memory”. You have to log out and in again for this change to take effect.

Further, your application has to be executed as an elevated process (“Run as administrator”) and the “SeLockMemoryPrivilege” must explicitely be enabled by your application. Example code on how to enable this privilege can be found in the “common/sys/alloc.cpp” file of Embree. Alternatively, Embree will try to enable this privilege when passing `enable_selockmemoryprivilege=1` to

`rtcNewDevice`. Further, huge pages have to get enabled in Embree by passing `hugepages=1` to `rtcNewDevice`.

When the system was running for a while, physical memory gets fragmented, which can slow down the allocation of huge pages significantly.

Huge Pages under Linux

Linux supports transparent huge pages and explicit huge pages. To enable transparent huge page support under Linux execute the following as root:

```
echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

When transparent huge pages are enabled, the kernel tries to merge 4k pages to 2MB pages when possible as a background job. See the following webpage for more information on transparent huge pages under Linux https://www.kernel.org/doc/Documentation/vm/transhuge.txt. In this mode each application, including your rendering application building on Embree, will automatically tend to use huge pages.

Using transparent huge pages the transitioning from 4k to 2MB pages might take some time. For that reason Embree also supports allocating 2MB pages directly when a huge page pool is configured. Such a pool can get configured by writing some number of huge pages to alloacte to `/proc/sys/vm/nr_overcommit_hugepages` as root user. E.g. to configure 2GB of adress space for huge page allocation, execute the following as root:

```
echo 1000 > /proc/sys/vm/nr_overcommit_hugepages
```

See the following webpage for more information on huge pages under Linux https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

Huge Pages under MacOSX

To use huge pages under MacOSX you have to pass `hugepages=1` to `rtcNewDevice` to enable that feature in Embree.

When the system was running for a while, physical memory gets quickly fragmented, and causes huge page allocations to fail. For this reason huge pages not very usefull under MacOSX in practise.

BVH Builder API

The Embree API exposes internal BVH builders to build BVHs with any desired node and leaf layout. To invoke the BVH builder you have to create a BVH object using the `rtcNewBVH` function and deleted again using the `rtcDeleteBVH` function.

```
RTCBVH rtcNewBVH(RTCDevice device);
void rtcDeleteBVH(RTCBVH bvh);
```

This BVH contains some builder state and fast node allocator. Some settings have to be passed to be BVH build function:

```
enum RTCBuildQuality
{
  RTC_BUILD_QUALITY_LOW = 0,     //!< build low quality BVH (good for dynamic scenes)
  RTC_BUILD_QUALITY_NORMAL = 1,  //!< build standard quality BVH
  RTC_BUILD_QUALITY_HIGH = 2,    //!< build high quality BVH
};
```

```
struct RTCBuildSettings
{
  unsigned size;               //!< size of this structure in bytes
  RTCBuildQuality quality;     //!< quality of BVH build
  unsigned maxBranchingFactor; //!< branching factor of BVH to build
  unsigned maxDepth;           //!< maximal depth of BVH to build
  unsigned sahBlockSize;       //!< blocksize for SAH heuristic
  unsigned minLeafSize;        //!< minimal size of a leaf
  unsigned maxLeafSize;        //!< maximal size of a leaf
  float travCost;              //!< estimated cost of one traversal step
  float intCost;               //!< estimated cost of one primitive intersection
  unsigned extraSpace;         //!< for spatial splitting we need extra space at end of primitive array
};
```

Some default values for the settings can be obtained using the `rtcDefaultBuildSettings` function. Using the `quality` setting, one can select between a faster low quality build which is good for dynamic scenes, and a standard quality build for static scenes. One can also specify the desired maximal branching factor of the BVH (`maxBranchingFactor` setting), the maximal depth the BVH should have (`maxDepth` setting), some power of 2 block size for the SAH heuristic (`sahBlockSize`), the minimal and maximal leaf size (`minLeafSize` and `maxLeafSize` setting), and the estimated cost of one traversal step and primitve intersection (`travCost` and `intCost` setting). To spatially split primitives in high quality mode, the builder needs some extra space at the end of the build primitive array. The amount of extra space can be passed using the `extraSpace` setting, and should be about the same size as there are primitives. The `size` member has always to be set to the size of the `RTCBuildSettings` structure in bytes.

Four callback functions have to get registered which are invoked during build to create BVH nodes (`RTCCreateNodeFunc`), set the pointers to all children (`RTCSetNodeChildrenFunc`), set the bounding boxes of all children (`RTCSetNodeBoundsFunc`), and to create a leaf node (`RTCCreateLeafFunc`).

```
typedef void* (*RTCCreateNodeFunc) (RTCThreadLocalAllocator allocator,
                                    size_t numChildren, void* userPtr);

typedef void  (*RTCSetNodeChildFunc) (void* nodePtr, void** childPtrs, size_t numChildren,
                                      void* userPtr);

typedef void  (*RTCSetNodeBoundsFunc) (void* nodePtr, const RTCBounds** bounds, size_t numChildren,
                                       void* userPtr);

typedef void* (*RTCCreateLeafFunc) (RTCThreadLocalAllocator allocator,
                                    const RTCBuildPrimitive* primitives, size_t numPrimitives,
                                    void* userPtr);

typedef void  (*RTCSplitPrimitiveFunc) (const RTCBuildPrimitive& prim,
    unsigned dim, float pos, RTCBounds& lbounds, RTCBounds& rbounds, void* userPtr);
```

The `RTCCreateNodeFunc` and `RTCCreateLeafFunc` type callbacks are passed a thread local allocator object that should be used for fast allocation of nodes using the `rtcThreadLocalAlloc` function.

```
void* rtcThreadLocalAlloc(RTCThreadLocalAllocator allocator, size_t bytes, size_t align);
```

We strongly recommend using this allocation mechanism, as alternative approaches like standard `malloc` can be over 10x slower. The allocator object passed to the create callbacks has to be used only inside the current thread.

The RTCCreateNodeFunc callback additionally gets passed the number of children for this node in the range from 2 to maxBranchingFactor (numChildren argument).

The RTCSetNodeChildFunc callback function, gets passed a pointer to the node as input (nodePtr argument), an array of pointers to the children (childPtrs argument), and the size of this array (numChildren argument).

The RTCSetNodeBoundsFunc callback function, get a pointer to the node as input (nodePtr argument), an array of pointers to the bounding boxes of the children (bounds argument), and the size of this array (numChildren argument).

The RTCCreateLeafFunc callback additionally get an array of primitives as input (primitives argument), and the size of this array (numPrimitives argument). The callback should read the geomID and primID members from the passed primitives to construct the leaf.

The RTCSplitPrimitiveFunc callback is invoked in high quality mode to split a primitive (prim argument) at some specified position (pos argument) and dimension (dim argument). The callback should return bounds of the clipped left and right part of the primitive (lbounds and rbounds arguments).

There is an optional progress callback function that can be used to get progress on the BVH build.

```
typedef void (*RTCBuildProgressFunc) (size_t N, void* userPtr);
```

This progress function is called with a number N of primitives the build is finished for. Accumulating over all invokations will sum up to the number of primitives passed to be BVH build function.

All callback functions are typically called from multiple threads, thus their implementation has to be thread safe.

All callback function get a user defined pointer (userPtr argument) as input which is provided to the rtcBuildBVH call. This pointer can be used to access the application scene object inside the callback functions.

The BVH build is invoked using the rtcBuildBVH function:

```
void* rtcBuildBVH(RTCBVH bvh,                                //!< BVH to build
                  const RTCBuildSettings& settings,          //!< settings for BVH builder
                  RTCBuildPrimitive* primitives,             //!< list of input primitives
                  size_t numPrimitives,                      //!< number of input primitives
                  RTCCreateNodeFunc createNode,              //!< creates a node
                  RTCSetNodeChildrenFunc setNodeChildren,    //!< sets pointer to a child
                  RTCSetNodeBoundsFunc setNodeBounds,        //!< sets bound of a child
                  RTCCreateLeafFunc createLeaf,              //!< creates a leaf
                  RTCSplitPrimitiveFunc splitPrimitive,      //!< splits a primitive into two halves
                  RTCBuildProgressFunc buildProgress         //!< used to report build progress
                  void* userPtr);                            //!< user pointer passed to callback functions
```

The function gets passed the BVH objects (bvh argument), the build settings to use (settings argument), the array of primitives (primitives argument) and its size (numPrimitives argument), the previously described callback function pointers, and a user defined pointer (userPtr argument) that is passed to all callback functions. The function pointer to the primitive split function (splitPrimitive argument) may be NULL, however, then no spatial splitting in high quality mode is possible. The function pointer used to report the build progress (buildProgress argument) is optional and may also be NULL.

For static scenes that do not require a further rtcBuildBVH call one should use the rtcMakeStatic function after the build which clears some internal data.

```
void rtcMakeStaticBVH(RTCBVH bvh);
```

Chapter 5
Embree Tutorials

Embree comes with a set of tutorials aimed at helping users understand how Embree can be used and extended. All tutorials exist in an ISPC and C++ version to demonstrate the two versions of the API. Look for files named `tutorialname_device.ispc` for the ISPC implementation of the tutorial, and files named `tutorialname_device.cpp` for the single ray C++ version of the tutorial. To start the C++ version use the `tutorialname` executables, to start the ISPC version use the `tutorialname_ispc` executables.

For all tutorials, you can select an initial camera using the `-vp` (camera position), `-vi` (camera look-at point), `-vu` (camera up vector), and `-fov` (vertical field of view) command line parameters:

```
./triangle_geometry -vp 10 10 10 -vi 0 0 0
```

You can select the initial windows size using the `-size` command line parameter, or start the tutorials in fullscreen using the `-fullscreen` parameter:

```
./triangle_geometry -size 1024 1024
./triangle_geometry -fullscreen
```

Implementation specific parameters can be passed to the ray tracing core through the `-rtcore` command line parameter, e.g.:

```
./triangle_geometry -rtcore verbose=2,threads=1,accel=bvh4.triangle1
```

The navigation in the interactive display mode follows the camera orbit model, where the camera revolves around the current center of interest. With the left mouse button you can rotate around the center of interest (the point initially set with `-vi`). Holding Control pressed while clicking the left mouse button rotates the camera around its location. You can also use the arrow keys for navigation.

You can use the following keys:

F1 Default shading
F2 Gray EyeLight shading
F3 Wireframe shading
F4 UV Coordinate visualization
F5 Geometry normal visualization
F6 Geometry ID visualization
F7 Geometry ID and Primitive ID visualization
F8 Simple shading with 16 rays per pixel for benchmarking.
F9 Switches to render cost visualization. Pressing again reduces brightness.
F10 Switches to render cost visualization. Pressing again increases brightness.
f Enters or leaves full screen mode.

c Prints camera parameters.
ESC Exits the tutorial.
q Exits the tutorial.

Triangle Geometry

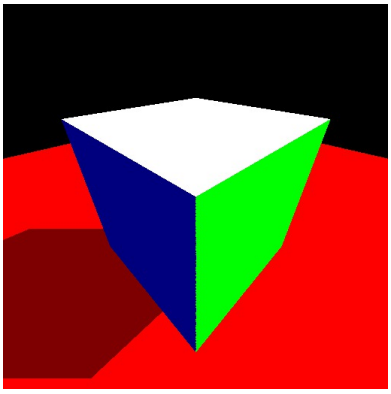

Figure 5.1

This tutorial demonstrates the creation of a static cube and ground plane using triangle meshes. It also demonstrates the use of the `rtcIntersect` and `rtcOccluded` functions to render primary visibility and hard shadows. The cube sides are colored based on the ID of the hit primitive.

Dynamic Scene

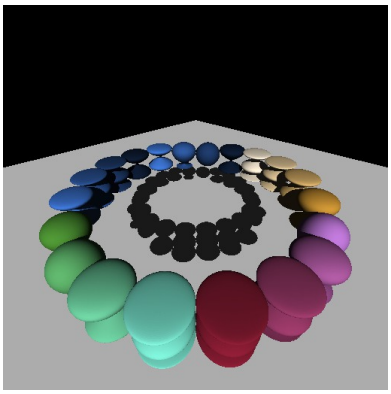

Figure 5.2

This tutorial demonstrates the creation of a dynamic scene, consisting of several deformed spheres. Half of the spheres use the `RTC_GEOMETRY_DEFORMABLE` flag, which allows Embree to use a refitting strategy for these spheres, the other half uses the `RTC_GEOMETRY_DYNAMIC` flag, causing a rebuild of their spatial data structure each frame. The spheres are colored based on the ID of the hit sphere geometry.

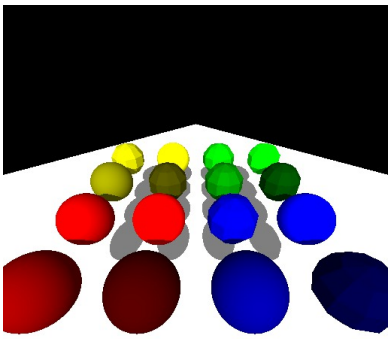

Figure 5.3

User Geometry

This tutorial shows the use of user defined geometry, to re-implement instancing and to add analytic spheres. A two level scene is created, with a triangle mesh as ground plane, and several user geometries, that instance other scenes with a small number of spheres of different kind. The spheres are colored using the instance ID and geometry ID of the hit sphere, to demonstrate how the same geometry, instanced in different ways can be distinguished.

Viewer

Figure 5.4

This tutorial demonstrates a simple OBJ viewer that traces primary visibility rays only. A scene consisting of multiple meshes is created, each mesh sharing the index and vertex buffer with the application. Demonstrated is also how to support additional per-vertex data, such as shading normals.

You need to specify an OBJ file at the command line for this tutorial to work:

```
./viewer -i model.obj
```

Stream Viewer

This tutorial demonstrates a simple OBJ viewer that demonstrates the use of ray streams. You need to specify an OBJ file at the command line for this tutorial to work:

```
./viewer_stream -i model.obj
```

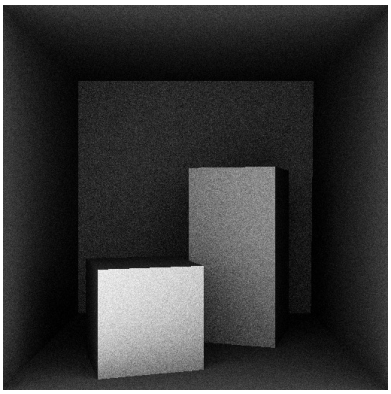

Figure 5.5

Instanced Geometry

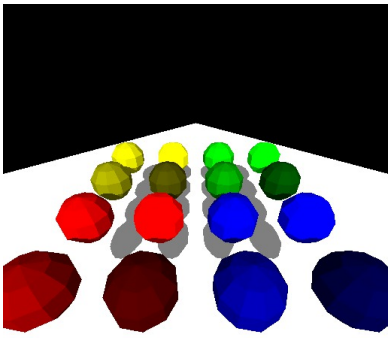

Figure 5.6

This tutorial demonstrates the in-build instancing feature of Embree, by instancing a number of other scenes build from triangulated spheres. The spheres are again colored using the instance ID and geometry ID of the hit sphere, to demonstrate how the same geometry, instanced in different ways can be distinguished.

Intersection Filter

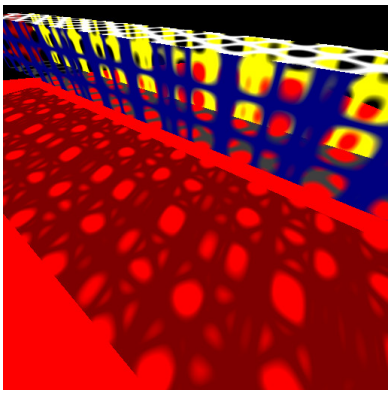

Figure 5.7

This tutorial demonstrates the use of filter callback functions to efficiently implement transparent objects. The filter function used for primary rays, lets the ray pass through the geometry if it is entirely transparent. Otherwise the shading loop handles the transparency properly, by potentially shooting secondary rays.

The filter function used for shadow rays accumulates the transparency of all surfaces along the ray, and terminates traversal if an opaque occluder is hit.

Pathtracer

Figure 5.8

This tutorial is a simple path tracer, building on the viewer tutorial.

You need to specify an OBJ file and light source at the command line for this tutorial to work:

```
./pathtracer -i model.obj -ambientlight 1 1 1
```

As example models we provide the "Austrian Imperial Crown" model by Martin Lubich and the "Asian Dragon" model from the Stanford 3D Scanning Repository.

crown.zip

asian_dragon.zip

To render these models execute the following:

```
./pathtracer -c crown/crown.ecs
./pathtracer -c asian_dragon/asian_dragon.ecs
```

Hair

Figure 5.9

This tutorial demonstrates the use of the hair geometry to render a hairball.

Bézier Curves

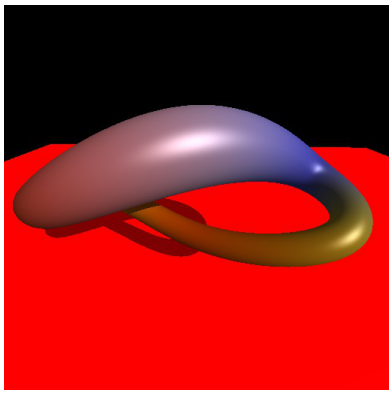

Figure 5.10

This tutorial demonstrates the use of the Bézier curve geometry.

Subdivision Geometry

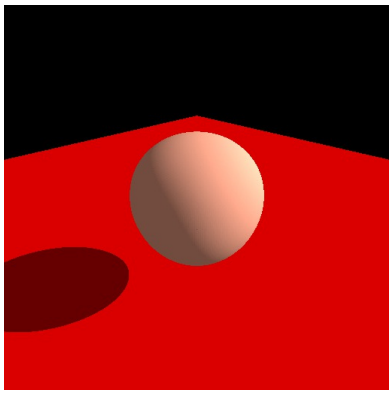

Figure 5.11

This tutorial demonstrates the use of Catmull Clark subdivision surfaces. Per default the edge tessellation level is set adaptively based on the distance to the camera origin. Embree currently supports three different modes for efficiently handling subdivision surfaces in various rendering scenarios. These three modes can be selected at the command line, e.g. `-lazy` builds internal per subdivision patch data structures on demand, `-cache` uses a small (per thread) tessellation cache for caching per patch data, and `-pregenerate` to generate and store most per patch data during the initial build process. The `cache` mode is most effective for coherent rays while providing a fixed memory footprint. The `pregenerate` modes is most effective for incoherent ray distributions while requiring more memory. The `lazy` mode works similar to the `pregenerate` mode but provides a middle ground in terms of memory consumption as it only builds and stores data only when the corresponding patch is accessed during the ray traversal. The `cache` mode is currently a bit more efficient at handling dynamic scenes where only the edge tessellation levels are changing per frame.

Displacement Geometry

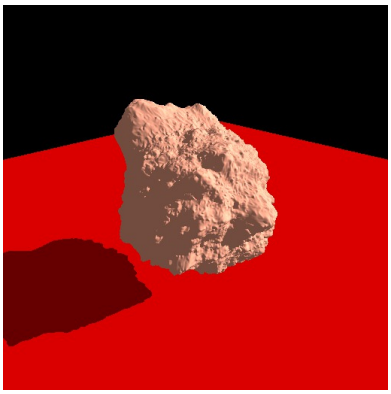

Figure 5.12

This tutorial demonstrates the use of Catmull Clark subdivision surfaces with procedural displacement mapping using a constant edge tessellation level.

Motion Blur Geometry

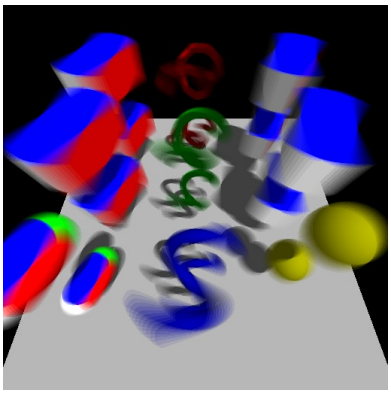

Figure 5.13

This tutorial demonstrates rendering of motion blur using the multi segment motion blur feature. Shown is motion blur of a triangle mesh, quad mesh, subdivision surface, line segments, hair geometry, bezier curves, instantiated triangle mesh where the instance moves, instantiated quad mesh where the instance and the quads move, and user geometry.

The number of time steps used can be configured using the `--time-steps <int>` and `--time-steps2 <int>` command line parameters, and the geometry can be rendered at a specific time using the the `--time <float>` command line parameter.

Interpolation

This tutorial demonstrates interpolation of user defined per-vertex data.

BVH Builder

This tutorial demonstrates how to use the templated hierarchy builders of Embree to build a bounding volume hierarchy with a user defined memory layout using a high quality SAH builder and very fast morton builder.

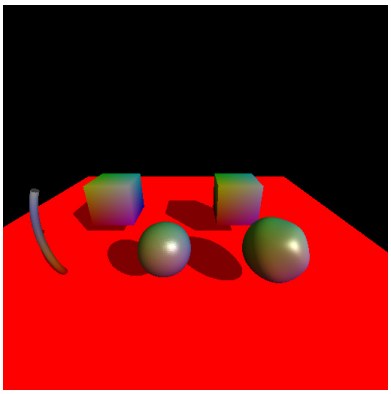

Figure 5.14

BVH Access

This tutorial demonstrates how to access the internal triangle acceleration structure build by Embree. Please be aware that the internal Embree data structures might change between Embree updates.

Find Embree

This tutorial demonstrates how to use the `FIND_PACKAGE` CMake feature to use an installed Embree. Under Linux and Mac OS X the tutorial finds the Embree installation automatically, under Windows the `embree_DIR` CMake variable has to be set to the following folder of the Embree installation: `C:\Program Files\Intel\Embree X.Y.Z\lib\cmake\embree-X.Y.Z`.