Intel® Embree
High Performance Ray Tracing Kernels

Version 4.0.1
March 9, 2023

Contents

Chapter 1

Intel® Embree Overview

Intel® Embree is a high-performance ray tracing library developed at Intel, which is released as open source under the Apache 2.0 license. Intel® Embree supports x86 CPUs under Linux, macOS, and Windows; ARM CPUs on macOS; as well as Intel® GPUs under Linux and Windows.

Intel® Embree targets graphics application developers to improve the performance of photo-realistic rendering applications. Embree is optimized towards production rendering, by putting focus on incoherent ray performance, high quality acceleration structure construction, a rich feature set, accurate primitive intersection, and low memory consumption.

Embree's feature set includes various primitive types such as triangles (as well quad and grids for lower memory consumption); Catmull-Clark subdivision surfaces; various types of curve primitives, such as flat curves (for distant views), round curves (for closeup views), and normal oriented curves, all supported with different basis functions (linear, Bézier, B-spline, Hermite, and Catmull Rom); point-like primitives, such as ray oriented discs, normal oriented discs, and spheres; user defined geometries with a procedural intersection function; multi-level instancing; filter callbacks invoked for any hit encountered; motion blur including multi-segment motion blur, deformation blur, and quaternion motion blur; and ray masking.

Intel® Embree contains ray tracing kernels optimized for the latest x86 processors with support for SSE, AVX, AVX2, and AVX-512 instructions, and uses runtime code selection to choose between these kernels. Intel® 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) as well as supports for dynamic scenes by implementing high-performance two-level spatial index structure construction algorithms.

Intel® Embree supports applications written with the Intel® Implicit SPMD Program Compiler (Intel® ISPC, `https://ispc.github.io/`) by providing an ISPC interface to the core ray tracing algorithms. This makes it possible to write a renderer that automatically vectorizes and leverages SSE, AVX, AVX2, and AVX-512 instructions.

Intel® Embree supports Intel GPUs through the SYCL open standard programming language. SYCL allows to write C++ code that can be run on various devices, such as CPUs and GPUs. Using Intel® Embree application developers can write a single source renderer that executes efficiently on CPUs and GPUs. Maintaining just one code base this way can significantly improve productivity and eliminate inconsistencies between a CPU and GPU version of the renderer. Embree supports GPUs based on the Xe HPG and Xe HPC microarchitecture, which support hardware accelerated ray tracing do deliver excellent levels of ray tracing performance.

1.1 Supported Platforms

Embree supports Windows (32-bit and 64-bit), Linux (64-bit), and macOS (64-bit). Under Windows, Linux and macOS x86 based CPUs are supported, while ARM CPUs are currently only supported under macOS (e.g. Apple M1). ARM support for Windows and Linux is experimental.

Embree supports Intel GPUs based on the Xe HPG microarchitecture (Intel® Arc™ GPU) under Linux and Windows and Xe HPC microarchitecture (Intel® Data Center GPU Flex Series and Intel® Data Center GPU Max Series) under Linux.

Currently the following products are supported and further products will get enabled soon:

- Intel® Arc™ A770 Graphics
- Intel® Arc™ A750 Graphics
- Intel® Arc™ A770M Graphics
- Intel® Arc™ A730M Graphics
- Intel® Arc™ A550M Graphics
- Intel® Data Center GPU Flex 170

The code compiles with the Intel® Compiler, Intel® oneAPI DPC++ Compiler, GCC, Clang, and the Microsoft Compiler. To use Embree on the GPU the Intel® oneAPI DPC++ Compiler must be used. Please see section Compiling Embree for details on tested compiler versions.

Embree requires at least an x86 CPU with support for SSE2 or an Apple M1 CPU.

1.2 Embree Support and Contact

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

For questions and feature requests 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.

1.3 Version History

1.3.1 Embree 4.0.1

- Improved performance for Tiger Lake, Comet Lake, Cannon Lake, Kaby Lake, and Skylake client CPUs by using 256 bit SIMD instructions by default.
- Fixed broken motion blur of RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE geometry type.
- Fixed bvh build retry issue for TBB 2020.3
- Added support for Intel® Data Center GPU Flex Series
- Fixed issue on systems without a SYCL platform.

1.3.2 Embree 4.0.0

- This Embree release adds support for Intel® Arc™ GPUs through SYCL.
- The SYCL support of Embree is in beta phase. Current functionality, quality, and GPU performance may not reflect that of the final product. Please read the documentation section "Embree SYCL Known Issues" for known limitations.

- Embree CPU support in this release as at Gold level, incorporating the same quality and performance as previous releases.
- A small number of API changes were required to get optimal experience and performance on the CPU and GPU. See documentation section “Upgrading from Embree 3 to Embree 4” for details.
- rtcIntersect and rtcOccluded function arguments changed slightly.
- RTCIntersectContext is renamed to RTCRayQuery context and most members moved to new RTCIntersectArguments and RTCOccludedArguments structures.
- rtcFilterIntersection and rtcFilterOcclusion API calls got replaced by rtcInvokeIntersectFilterFromGeometry and rtcInvokeOccludedFilterFromGeometry API calls.
- rtcSetGeometryEnableFilterFunctionFromArguments enables argument filter functions for some geometry.
- RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER ray query flag enables argument filter functions for each geometry.
- User geometry callbacks have to return if a valid hit was found.
- Ray masking is enabled by default now as required by most users.
- The default ray mask for geometries got changed from 0xFFFFFFFF to 0x1.
- Removed ray stream API as rarely used with minimal performance benefits over packet tracing.
- Introduced rtcForwardIntersect/rtcForwardOccluded API calls to trace tail recursive rays from user geometry callback.
- The rtcGetGeometryUserDataFromScene API call got added to be used in SYCL code.
- Added support for user geometry callback function pointer passed through ray query context
- Feature flags enable reducing code complexity for optimal performance on the GPU.
- Fixed compilation issues for ARM AArch64 processor under Linux.
- Setting default frequency level to SIMD256 for ARM on all platforms. This allows using double pumped NEON execution by enabling EMBREE_ISA_NEON2X in cmake under Linux.
- Fixed missing end caps of motion blurred line segments.
- EMBREE_ISPC_SUPPORT is turned OFF by default.
- Embree drops support of the deprecated Intel(R) Compiler. It is replaced by the Intel(R) oneAPI DPC++/C++ Compiler on Windows and Linux and the Intel(R) C++ Classic Compiler on MacOS (latest tested versions is 2023.0.0).

1.3.3 Embree 3.13.5

- Fixed bug in bounding flat Catmull Rom curves of subdivision level 4.
- Improved self intersection avoidance for RTC_GEOMETRY_TYPE_DISC_POINT geometry type. Intersections are skipped if the ray origin lies inside the sphere defined by the point primitive. Self intersection avoidance can get disabled at compile time using the EMBREE_DISC_POINT_SELF_INTERSECTION_AVOIDANCE cmake option.
- Fixed spatial splitting for non-planar quads.

1.3.4 Embree 3.13.4

- Using 8-wide BVH and double pumped NEON instructions on Apple M1 gives 8% performance boost.
- Fixed binning related crash in SAH BVH builder.
- Added EMBREE_TBB_COMPONENT cmake option to define the component/library name of Intel® TBB (default: tbb).

- Embree supports now Intel® oneAPI DPC++/C++ Compiler 2022.0.0

1.3.5 Embree 3.13.3

- Invalid multi segment motion blurred normal oriented curves are properly excluded from BVH build.
- Fixing issue with normal oriented curve construction when center curve curvature is very large. Due to this change normal oriented curve shape changes slightly.
- Fixed crash caused by disabling a geometry and then detaching it from the scene.
- Bugfix in emulated ray packet intersection when EMBREE_RAY_PACKETS is turned off.
- Bugfix for linear quaternion interpolation fallback.
- Fixed issues with spaces in path to Embree build folder.
- Some fixes to compile Embree in SSE mode using WebAssembly.
- Bugfix for occlusion rays with grids and ray packets.
- We do no longer provide installers for Windows and macOS, please use the ZIP files instead.
- Upgrading to Intel® ISPC 1.17.0 for release build.
- Upgrading to Intel® oneTBB 2021.5.0 for release build.

1.3.6 Embree 3.13.2

- Avoiding spatial split positions that are slightly out of geometry bounds.
- Introduced rtcGetGeometryThreadSafe function, which is a thread safe version of rtcGetGeometry.
- Using more accurate rcp implementation.
- Bugfix to rare corner case of high quality BVH builder.

1.3.7 Embree 3.13.1

- Added support for Intel® ISPC ARM target.
- Releases upgrade to Intel® TBB 2021.3.0 and Intel® ISPC 1.16.1

1.3.8 Embree 3.13.0

- Added support for Apple M1 CPUs.
- RTC_SUBDIVISION_MODE_NO_BOUNDARY now works properly for non-manifold edges.
- CMake target ‘uninstall’ is not defined if it already exists.
- Embree no longer reads the .embree3 config files, thus all configuration has to get passed through the config string to rtcNewDevice.
- Releases upgrade to Intel® TBB 2021.2.0 and Intel® ISPC 1.15.0
- Intel® TBB dll is automatically copied into build folder after build on windows.

1.3.9 Embree 3.12.2

- Fixed wrong uv and Ng for grid intersector in robust mode for AVX.
- Removed optimizations for Knights Landing.
- Upgrading release builds to use Intel® oneTBB 2021.1.1

1.3.10 Embree 3.12.1

- Changed default frequency level to SIMD128 for Skylake, Cannon Lake, Comet Lake and Tiger Lake CPUs. This change typically improves perfor-

mance for renderers that just use SSE by maintaining higher CPU frequencies. In case your renderer is AVX optimized you can get higher ray tracing performance by configuring the frequency level to simd256 through passing frequency_level=simd256 to rtcNewDevice.

1.3.11 Embree 3.12.0

- Added linear cone curve geometry support. In this mode a real geometric surface for curves with linear basis is rendered using capped cones. They are discontinuous at edge boundaries.
- Enabled fast two level builder for instances when low quality build is requested.
- Bugfix for BVH build when geometries got disabled.
- Added EMBREE_BACKFACE_CULLING_CURVES cmake option. This allows for a cheaper round linear curve intersection when correct internal tracking and back hits are not required. The new cmake option defaults to OFF.
- User geometries with invalid bounds with lower>upper in some dimension will be ignored.
- Increased robustness for grid interpolation code and fixed returned out of range u/v coordinates for grid primitive.
- Fixed handling of motion blur time range for sphere, discs, and oriented disc geometries.
- Fixed missing model data in releases.
- Ensure compatibility to newer versions of Intel® oneTBB.
- Motion blur BVH nodes no longer store NaN values.

1.3.12 Embree 3.11.0

- Round linear curves now automatically check for the existence of left and right connected segments if the flags buffer is empty. Left segments exist if the segment(id-1) + 1 == segment(id) and similarly for right segments.
- Implemented the min-width feature for curves and points, which allows to increase the radius in a distance dependent way, such that the curve or points thickness is n pixels wide.
- Round linear curves are closed now also at their start.
- Embree no longer supports Visual Studio 2013 starting with this release.
- Bugfix in subdivision tessellation level assignment for non-quad base primitives
- Small meshes are directly added to top level build phase of two-level builder to reduce memory consumption.
- Enabled fast two level builder for user geometries when low quality build is requested.

1.3.13 Embree 3.10.0

- Added EMBREE_COMPACT_POLYS CMake option which enables double indexed triangle and quad leaves to reduce memory consumption in compact mode by an additional 40% at about 15% performance impact. This new mode is disabled by default.
- Compile fix for Intel® oneTBB 2021.1-beta05
- Releases upgrade to Intel® TBB 2020.2
- Compile fix for Intel® ISPC v1.13.0
- Adding RPATH to libembree.so in releases
- Increased required CMake version to 3.1.0
- Made instID member for array of pointers ray stream layout optional again.

1.3.14 Embree 3.9.0

- Added round linear curve geometry support. In this mode a real geometric surface for curves with linear basis is rendered using capped cones with spherical filling between the curve segments.
- Added rtcGetSceneDevice API function, that returns the device a scene got created in.
- Improved performance of round curve rendering by up to 1.8x.
- Bugfix to sphere intersection filter invocation for back hit.
- Fixed wrong assertion that triggered for invalid curves which anyway get filtered out.
- RelWithDebInfo mode no longer enables assertions.
- Fixed an issue in FindTBB.cmake that caused compile error with Debug build under Linux.
- Embree releases no longer provide RPMs for Linux. Please use the RPMs coming with the package manager of your Linux distribution.

1.3.15 Embree 3.8.0

- Added collision detection support for user geometries (see rtcCollide API function)
- Passing geomID to user geometry callbacks.
- Bugfix in AVX512VL codepath for rtcIntersect1
- For sphere geometries the intersection filter gets now invoked for front and back hit.
- Fixed some bugs for quaternion motion blur.
- RTCRayQueryContext always non-const in Embree API
- Made RTCHit aligned to 16 bytes in Embree API

1.3.16 New Features in Embree 3.7.0

- Added quaternion motion blur for correct interpolation of rotational transformations.
- Fixed wrong bounding calculations when a motion blurred instance did instantiate a motion blurred scene.
- In robust mode the depth test consistently uses tnear <= t <= tfar now in order to robustly continue traversal at a previous hit point in a way that guarantees reaching all hits, even hits at the same place.
- Fixed depth test in robust mode to be precise at tnear and tfar.
- Added next_hit tutorial to demonstrate robustly collecting all hits along a ray using multiple ray queries.
- Implemented robust mode for curves. This has a small performance impact but fixes bounding problems with flat curves.
- Improved quality of motion blur BVH by using linear bounds during binning.
- Implemented issue with motion blur builder where number of time segments for SAH heuristic were counted wrong due to some numerical issues.
- Fixed an accuracy issue with rendering very short fat curves.
- rtcCommitScene can now get called during rendering from multiple threads to lazily build geometry. When Intel® TBB is used this causes a much lower overhead than using rtcJoinCommitScene.
- Geometries can now get attached to multiple scenes at the same time, which simplifies mapping general scene graphs to API.
- Updated to Intel® TBB 2019.9 for release builds.
- Fixed a bug in the BVH builder for Grid geometries.
- Added macOS Catalina support to Embree releases.

1.3.17 New Features in Embree 3.6.1

- Restored binary compatibility between Embree 3.6 and 3.5 when single-level instancing is used.
- Fixed bug in subgrid intersector
- Removed point query alignment in Intel® ISPC header

1.3.18 New Features in Embree 3.6

- Added Catmull-Rom curve types.
- Added support for multi-level instancing.
- Added support for point queries.
- Fixed a bug preventing normal oriented curves being used unless timesteps were specified.
- Fixed bug in external BVH builder when configured for dynamic build.
- Added support for new config flag "user_threads=N" to device initialization which sets the number of threads used by Intel® TBB but created by the user.
- Fixed automatic vertex buffer padding when using rtcSetNewGeometry API function.

1.3.19 New Features in Embree 3.5.2

- Added EMBREE_API_NAMESPACE cmake option that allows to put all Embree API functions inside a user defined namespace.
- Added EMBREE_LIBRARY_NAME cmake option that allows to rename the Embree library.
- When Embree is compiled as static library, EMBREE_STATIC_LIB has no longer to get defined before including the Embree API headers.
- Added CPU frequency_level device configuration to allow an application to specify the frequency level it wants to run on. This forces Embree to not use optimizations that may reduce the CPU frequency below that level. By default Embree is configured to the the AVX-heavy frequency level, thus if the application uses solely non-AVX code, configuring the Embree device with "frequency_level=simd128" may give better performance.
- Fixed a bug in the spatial split builder which caused it to fail for scenes with more than 2^24 geometries.

1.3.20 New Features in Embree 3.5.1

- Fixed ray/sphere intersector to work also for non-normalized rays.
- Fixed self intersection avoidance for ray oriented discs when non-normalized rays were used.
- Increased maximal face valence for subdiv patch to 64 and reduced stack size requirement for subdiv patch evaluation.

1.3.21 New Features in Embree 3.5.0

- Changed normal oriented curve definition to fix waving artefacts.
- Fixed bounding issue for normal oriented motion blurred curves.
- Fixed performance issue with motion blurred point geometry.
- Fixed generation of documentation with new pandoc versions.

1.3.22 New Features in Embree 3.4.0

- Added point primitives (spheres, ray-oriented discs, normal-oriented discs).
- Fixed crash triggered by scenes with only invalid primitives.

- Improved robustness of quad/grid-based intersectors.
- Upgraded to Intel® TBB 2019.2 for release builds.

1.3.23 New Features in Embree 3.3.0

- Added support for motion blur time range per geometry. This way geometries can appear and disappear during the camera shutter and time steps do not have to start and end at camera shutter interval boundaries.
- Fixed crash with pathtracer when using –triangle-sphere command line.
- Fixed crash with pathtracer when using –shader ao command line.
- Fixed tutorials showing a black window on macOS 10.14 until moved.

1.3.24 New Features in Embree 3.2.4

- Fixed compile issues with ICC 2019.
- Released ZIP files for Windows are now provided in a version linked against Visual Studio 2013 and Visual Studio 2015.

1.3.25 New Features in Embree 3.2.3

- Fixed crash when using curves with RTC_SCENE_FLAG_DYNAMIC combined with RTC_BUILD_QUALITY_MEDIUM.

1.3.26 New Features in Embree 3.2.2

- Fixed intersection distance for unnormalized rays with line segments.
- Removed libmmd.dll dependency in release builds for Windows.
- Fixed detection of AppleClang compiler under MacOSX.

1.3.27 New Features in Embree 3.2.1

- Bugfix in flat mode for hermite curves.
- Added EMBREE_CURVE_SELF_INTERSECTION_AVOIDANCE_FACTOR cmake option to control self intersection avoidance for flat curves.
- Performance fix when instantiating motion blurred scenes. The application should best use two (or more) time steps for an instance that instantiates a motion blurred scene.
- Fixed AVX512 compile issue with GCC 6.1.1.
- Fixed performance issue with rtcGetGeometryUserData when used during rendering.
- Bugfix in length of derivatives for grid geometry.
- Added BVH8 support for motion blurred curves and lines. For some workloads this increases performance by up to 7%.
- Fixed rtcGetGeometryTransform to return the local to world transform.
- Fixed bug in multi segment motion blur that caused missing of perfectly axis aligned geometry.
- Reduced memory consumption of small scenes by 4x.
- Reduced temporal storage of grid builder.

1.3.28 New Features in Embree 3.2.0

- Improved watertightness of robust mode.
- Line segments, and other curves are now all contained in a single BVH which improves performance when these are both used in a scene.
- Performance improvement of up to 20% for line segments.
- Bugfix to Embree2 to Embree3 conversion script.
- Added support for Hermite curve basis.

- Semantics of normal buffer for normal oriented curves has changed to simplify usage. Please see documentation for details.
- Using GLFW and imgui in tutorials.
- Fixed floating point exception in static variable initialization.
- Fixed invalid memory access in rtcGetGeometryTransform for non-motion blur instances.
- Improved self intersection avoidance for flat curves. Transparency rays with tnear set to previous hit distance do not need curve radius based self intersection avoidance as same hit is calculated again. For this reason self intersection avoidance is now only applied to ray origin.

1.3.29 New Features in Embree 3.1.0

- Added new normal-oriented curve primitive for ray tracing of grass-like structures.
- Added new grid primitive for ray tracing tessellated and displaced surfaces in very memory efficient manner.
- Fixed bug of ribbon curve intersector when derivative was zero.
- Installing all static libraries when EMBREE_STATIC_LIB is enabled.
- Added API functions to access topology of subdivision mesh.
- Reduced memory consumption of instances.
- Improved performance of instances by 8%.
- Reduced memory consumption of curves by up to 2x.
- Up to 5% higher performance on AVX-512 architectures.
- Added native support for multiple curve basis functions. Internal basis conversions are no longer performed, which saves additional memory when multiple bases are used.
- Fixed issue with non thread safe local static variable initialization in VS2013.
- Bugfix in rtcSetNewGeometry. Vertex buffers did not get properly overallocated.
- Replaced ImageMagick with OpenImageIO in the tutorials.

1.3.30 New Features in Embree 3.0.0

- Switched to a new version of the API which provides improved flexibility but is not backward compatible. Please see “Upgrading from Embree 2 to Embree 3” section of the documentation for upgrade instructions. In particular, we provide a Python script that performs most of the transition work.
- User geometries inside an instanced scene and a top-level scene no longer need to handle the instID field of the ray differently. They both just need to copy the context.instID into the ray.instID field.
- Support for context filter functions that can be assigned to a ray query.
- User geometries can now invoke filter functions using the rtcFilterIntersection and rtcFilterOcclusion calls.
- Higher flexibility through specifying build quality per scene and geometry.
- Geometry normal uses commonly used right-hand rule from now on.
- Added self-intersection avoidance to ribbon curves and lines. Applications do not have to implement self-intersection workarounds for these primitive types anymore.
- Added support for 4 billion primitives in a single scene.
- Removed the RTC_MAX_USER_VERTEX_BUFFERS and RTC_MAX_INDEX_BUFFERS limitations.
- Reduced memory consumption by 192 bytes per instance.
- Fixed some performance issues on AVX-512 architectures.

- Individual Contributor License Agreement (ICLA) and Corporate Contributor License Agreement (CCLA) no longer required to contribute to the project.

1.3.31 New Features in Embree 2.17.5

- Improved watertightness of robust mode.
- Fixed floating point exception in static variable initialization.
- Fixed AVX512 compile issue with GCC 6.1.1.

1.3.32 New Features in Embree 2.17.4

- Fixed AVX512 compile issue with GCC 7.
- Fixed issue with not thread safe local static variable initialization in VS2013.
- Fixed bug in the 4 and 8-wide packet intersection of instances with multi-segment motion blur on AVX-512 architectures.
- Fixed bug in rtcOccluded4/8/16 when only AVX-512 ISA was enabled.

1.3.33 New Features in Embree 2.17.3

- Fixed GCC compile warning in debug mode.
- Fixed bug of ribbon curve intersector when derivative was zero.
- Installing all static libraries when EMBREE_STATIC_LIB is enabled.

1.3.34 New Features in Embree 2.17.2

- Made BVH build of curve geometry deterministic.

1.3.35 New Features in Embree 2.17.1

- Improved performance of occlusion ray packets by up to 50%.
- Fixed detection of Clang for CMake 3 under MacOSX
- Fixed AVX code compilation issue with GCC 7 compiler caused by explicit use of vzeroupper intrinsics.
- Fixed an issue where Clang address sanitizer reported an error in the internal tasking system.
- Added fix to compile on 32 bit Linux distribution.
- Fixed some wrong relative include paths in Embree.
- Improved performance of robust single ray mode by 5%.
- Added EMBREE_INSTALL_DEPENDENCIES option (default OFF) to enable installing of Embree dependencies.
- Fixed performance regression for occlusion ray streams.
- Reduced temporary memory requirements of BVH builder for curves and line segments.
- Fixed performance regression for user geometries and packet ray tracing.
- Fixed bug where wrong closest hit was reported for very curvy hair segment.

1.3.36 New Features in Embree 2.17.0

- Improved packet ray tracing performance for coherent rays by 10-60% (requires RTC_INTERSECT_COHERENT flag).
- Improved ray tracing performance for incoherent rays on AVX-512 architectures by 5%.
- Improved ray tracing performance for streams of incoherent rays by 5-15%.
- Fixed tbb_debug.lib linking error under Windows.

- Fast coherent ray stream and packet code paths now also work in robust mode.
- Using less aggressive prefetching for large BVH nodes which results in 1-2% higher ray tracing performance.
- Precompiled binaries have stack-protector enabled, except for traversal kernels. BVH builders can be slightly slower due to this change. If you want stack-protectors disabled please turn off EMBREE_STACK_PROTECTOR in cmake and build the binaries yourself.
- When enabling ISAs individually, the 8-wide BVH was previously only available when the AVX ISA was also selected. This issue is now fixed, and one can enable only AVX2 and still get best performance by using an 8-wide BVH.
- Fixed rtcOccluded1 and rtcOccluded1Ex API functions which were broken in Intel® ISPC.
- Providing MSI installer for Windows.

1.3.37 New Features in Embree 2.16.5

- Bugfix in the robust triangle intersector that rarely caused NaNs.
- Fixed bug in hybrid traversal kernel when BVH leaf was entered with no active rays. This rarely caused crashes when used with instancing.
- Fixed bug introduced in Embree 2.16.2 which caused instancing not to work properly when a smaller than the native SIMD width was used in ray packet mode.
- Fixed bug in the curve geometry intersector that caused rendering artefacts for Bézier curves with p0=p1 and/or p2=p3.
- Fixed bug in the curve geometry intersector that caused hit results with NaNs to be reported.
- Fixed masking bug that caused rare cracks in curve geometry.
- Enabled support for SSE2 in precompiled binaries again.

1.3.38 New Features in Embree 2.16.4

- Bugfix in the ribbon intersector for hair primitives. Non-normalized rays caused wrong intersection distance to be reported.

1.3.39 New Features in Embree 2.16.3

- Increased accuracy for handling subdivision surfaces. This fixes cracks when using displacement mapping but reduces performance at irregular vertices.
- Fixed a bug where subdivision geometry was not properly updated when modifying only the tessellation rate and vertex array.

1.3.40 New Features in Embree 2.16.2

- Fixed bug that caused NULL ray query context in intersection filter when instancing was used.
- Fixed an issue where uv's where outside the triangle (or quad) for very small triangles (or quads). In robust mode we improved the uv calculation to avoid that issue, in fast mode we accept that inconsistency for better performance.
- Changed UV encoding for non-quad subdivision patches to allow a sub-patch UV range of `[-0.5,1.5[`. Using this new encoding one can use finite differences to calculate derivatives if required. Please adjust your code in case you rely on the old encoding.

1.3.41 New Features in Embree 2.16.1

- Workaround for compile issues with Visual Studio 2017
- Fixed bug in subdiv code for static scenes when using tessellation levels larger than 50.
- Fixed low performance when adding many geometries to a scene.
- Fixed high memory consumption issue when using instances in dynamic scene (by disabling two level builder for user geometries and instances).

1.3.42 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 AVX-512 architectures by 5-10%.
- Improved packet/hybrid ray tracing performance for incoherent rays on AVX-512 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.

1.3.43 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.

1.3.44 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.

1.3.45 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.

1.3.46 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 Intel® 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 Intel® TBB is highly recommended).
- Updated to Intel® 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.

1.3.47 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.

1.3.48 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.

1.3.49 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.

1.3.50 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.

1.3.51 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 Intel® TBB 4.4 update 2).
- Improved performance through huge page support under linux.

1.3.52 New Features in Embree 2.7.1

- Internal tasking system supports cancellation of build operations.
- Intel® 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.

1.3.53 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 tessellation 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.

1.3.54 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.

1.3.55 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)

1.3.56 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

1.3.57 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™

1.3.58 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 Intel® TBB as default tasking system on Xeon to get even faster hierarchy build times and better integration for applications that also use Intel® TBB
- `rtcCommit` can get called from multiple Intel® TBB threads to join the hierarchy build operations

1.3.59 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)

1.3.60 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)

1.3.61 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)

1.3.62 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)

1.3.63 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)

1.3.64 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)

1.3.65 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)

1.3.66 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® Implicit SPMD Program Compiler (Intel® ISPC)
- Instantiation and fast BVH reconstruction
- Example photo-realistic rendering engine for both C++ and Intel® ISPC

Chapter 2

Installation of Embree

2.1 Windows Installation

Embree linked against Visual Studio 2015 are provided as a ZIP file embree-4.0.1.x64.vc14.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.

2.2 Linux Installation

The Linux version of Embree is also delivered as a `tar.gz` file: embree-4.0.1.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 set up the environment properly:

```
tar xzf embree-4.0.1.x86_64.linux.tar.gz
source embree-4.0.1.x86_64.linux/embree-vars.sh
```

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

2.3 macOS Installation

The macOS version of Embree is also delivered as a ZIP file: embree-4.0.1.x86_64.macosx.zip. 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 set up the environment properly:

```
unzip embree-4.0.1.x64.macosx.zip    source embree-4.0.1.x64.macosx/embree-vars.sh
```

If you want to ship Embree with your application, please use the Embree library of the provided ZIP file. The library name of that Embree library is of the form `@rpath/libembree.4.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 where Embree (and TBB) can be found, e.g. `@loader_path/../lib`.

2.4 Building Embree Applications

The most convenient way to build an Embree application is through CMake. Just let CMake find your unpacked Embree package using the `FIND_PACKAGE` function inside your `CMakeLists.txt` file:

```
FIND_PACKAGE(embree 4 REQUIRED)
```

For CMake to properly find Embree you need to set the `embree_DIR` variable to the folder containing the `embree_config.cmake` file. You might also have to set the `TBB_DIR` variable to the path containing `TBB-config.cmake` of a local TBB install, in case you do not have TBB installed globally on your system, e.g:

```
cmake -D embree_DIR=path_to_embree_package/lib/cmake/embree-4.0.1/ \
      -D TBB_DIR=path_to_tbb_package/lib/cmake/tbb/ \
      ..
```

The `FIND_PACKAGE` function will create an `embree` target that you can add to your target link libraries:

```
TARGET_LINK_LIBRARIES(application embree)
```

For a full example on how to build an Embree application please have a look at the `minimal` tutorial provided in the `src` folder of the Embree package and also the contained `README.txt` file.

2.5 Building Embree SYCL Applications

Building Embree SYCL applications is also best done using CMake. Please first get some compatible SYCL compiler and setup the environment as decribed in sections Linux SYCL Compilation and Windows SYCL Compilation.

Also perform the setup steps from the previous Building Embree Applications section.

Please also have a look at the Minimal tutorial that is provided with the Embree release, for an example how to build a simple SYCL application using CMake and Embree.

To properly compile your SYCL application you have to add additional SYCL compile flags for each C++ file that contains SYCL device side code or kernels as described next.

2.5.1 JIT Compilation

We recommend using just in time compilation (JIT compilation) together with SYCL JIT caching to compile Embree SYCL applications. For JIT compilation add these options to the compilation phase of all C++ files that contain SYCL code:

```
-fsycl -Xclang -fsycl-allow-func-ptr -fsycl-targets=spir64
```

These options enable SYCL two phase compilation (`-fsycl` option), enable function pointer support (`-Xclang -fsycl-allow-func-ptr` option), and just in time (JIT) compilation only (`-fsycl-targets=spir64` option).

The following link options have to get added to the linking stage of your application when using just in time compilation:

```
-fsycl -fsycl-targets=spir64
```

For a full example on how to build an Embree SYCL application please have a look at the SYCL version of the `minimal` tutorial provided in the `src` folder of the Embree package and also the contained `README.txt` file.

Please have a look at the Compiling Embree section on how to create an Embree package from sources if required.

2.5.2 AOT Compilation

Ahead of time compilation (AOT compilation) allows to speed up first application start up time as device binaries are precompiled. We do not recommend using AOT compilation as it does not allow the usage of specialization constants to reduce code complexity.

For ahead of time compilation add these compile options to the compilation phase of all C++ files that contain SYCL code:

```
-fsycl -Xclang -fsycl-allow-func-ptr -fsycl-targets=spir64_gen
```

These options enable SYCL two phase compilation (`-fsycl` option), enable function pointer support (`-Xclang -fsycl-allow-func-ptr` option), and ahead of time (AOT) compilation (`-fsycl-targets=spir64_gen` option).

The following link options have to get added to the linking stage of your application when compiling ahead of time for Xe HPG devices:

```
-fsycl -fsycl-targets=spir64_gen
-Xsycl-target-backend=spir64_gen "-device XE_HPG_CORE"
```

This in particular configures the devices for AOT compilation to `XE_HPG_CORE`.

To get a list of all device supported by AOT compilation look at the help of the device option in ocloc tool:

```
ocloc compile --help
```

Chapter 3
Compiling Embree

We recommend using the prebuild Embree packages from https://github.com/embree/embree/releases. If you need to compile Embree yourself you need to use CMake as described in the following.

Do not enable fast-math optimizations in your compiler as this mode is not supported by Embree.

3.1 Linux and macOS

To compile Embree you need a modern C++ compiler that supports C++11. Embree is tested with the following compilers:

Linux

- Intel® oneAPI DPC++/C++ Compiler 2023.0.0
- oneAPI DPC++/C++ Compiler 2022-12-14
- Clang 5.0.0
- Clang 4.0.0
- GCC 10.0.1 (Fedora 32) AVX512 support
- GCC 8.3.1 (Fedora 28) AVX512 support
- GCC 7.3.1 (Fedora 27) AVX2 support
- GCC 7.3.1 (Fedora 26) AVX2 support
- GCC 6.4.1 (Fedora 25) AVX2 support
- Intel® Implicit SPMD Program Compiler 1.18.1
- Intel® Implicit SPMD Program Compiler 1.17.0
- Intel® Implicit SPMD Program Compiler 1.16.1
- Intel® Implicit SPMD Program Compiler 1.15.0
- Intel® Implicit SPMD Program Compiler 1.14.1
- Intel® Implicit SPMD Program Compiler 1.13.0
- Intel® Implicit SPMD Program Compiler 1.12.0

macOS x86

- Intel® C++ Classic Compiler 2023.0.0
- Apple Clang 12.0.5 (macOS 11.7.1)

macOS M1

- Apple Clang 12.0.5 (macOS 11.7.1)

Embree supports using the Intel® Threading Building Blocks (TBB) as the tasking system. For performance and flexibility reasons we recommend using 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® Implicit SPMD Program Compiler (Intel® ISPC), which allows straightforward parallelization of an entire renderer. If you want to use Intel® ISPC then you can enable `EMBREE_ISPC_SUPPORT` in CMake. Download and install the Intel® ISPC binaries from ispc.github.io. After installation, put the path to `ispc` permanently into your `PATH` environment variable or you set the `EMBREE_ISPC_EXECUTABLE` variable to point at the ISPC executable during CMake configuration.

You additionally have to install CMake 3.1.0 or higher and the developer version of GLFW version 3.

Under macOS, all these dependencies can be installed using MacPorts:

```
sudo port install cmake tbb-devel glfw-devel
```

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
sudo yum install tbb-devel
sudo yum install glfw-devel
```

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 libglfw3-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 Clang compiler instead of the default GCC on most Linux machines (`g++` and `gcc`), execute

```
cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang ..
```

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 -j 8
```

The executables will be generated inside the build folder. We recommend installing 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
```

You can also create an Embree package using the following command:

```
make package
```

Please see the Building Embree Applications section on how to build your application with such an Embree package.

3.2 Linux SYCL Compilation

There are two options to compile Embree with SYCL support: The open source "oneAPI DPC++ Compiler" or the "Intel(R) oneAPI DPC++/C++ Compiler". Other SYCL compilers are not supported.

The "oneAPI DPC++ Compiler" is more up-to-date than the "Intel(R) oneAPI DPC++/C++ Compiler" but less stable. The current tested version of the "oneAPI DPC++ compiler is

- oneAPI DPC++ Compiler 2022-12-14

The compiler can be downloaded and simply extracted. The oneAPI DPC++ compiler 2022-12-14 can be set up executing the following command in a Linux (bash) shell:

```
wget https://github.com/intel/llvm/releases/download/sycl-nightly%2F20221214/dpcpp-compiler.tar.gz
tar xzf dpcpp-compiler.tar.gz
source ./dpcpp_compiler/startup.sh
```

The `startup.sh` script will put `clang++` and `clang` from the oneAPI DPC++ Compiler into your path.

Please also install all Linux packages described in the previous section.

Now, you can configure Embree using CMake by executing the following command in the Embree root directory:

```
cmake -B build \
      -DCMAKE_CXX_COMPILER=clang++ \
      -DCMAKE_C_COMPILER=clang \
      -DEMBREE_SYCL_SUPPORT=ON
```

This will create a directory `build` to use as the CMake build directory, configure the usage of the oneAPI DPC++ Compiler, and turn on SYCL support through `EMBREE_SYCL_SUPPORT=ON`.

Alternatively, you can download and run the installer of the

- Intel(R) oneAPI DPC++/C++ Compiler 2023.0.0.

After installation, you can set up the compiler by sourcing the `vars.sh` script in the `env` directory of the compiler install directory, for example,

```
source /opt/intel/oneAPI/compiler/2023.0.0/env/vars.sh
```

This script will put the `icpx` and `icx` compiler executables from the Intel(R) oneAPI DPC++/C++ Compiler in your path.

Now, you can configure Embree using CMake by executing the following command in the Embree root directory:

```
cmake -B build \
      -DCMAKE_CXX_COMPILER=icpx \
      -DCMAKE_C_COMPILER=icx \
      -DEMBREE_SYCL_SUPPORT=ON
```

More information about setting up the Intel(R) oneAPI DPC++/C++ compiler can be found in the Development Reference Guide. Please note, that the Intel(R) oneAPI DPC++/C++ compiler requires at least CMake version 3.20.5 on Linux.

Independent of the DPC++ compiler choice, you can now build Embree using

```
cmake --build build -j 8
```

The executables will be generated inside the build folder. The executable names of the SYCL versions of the tutorials end with `_sycl`.

3.2.1 Linux Graphics Driver Installation

To run the SYCL code you need to install the latest GPGPU drivers for your Intel Xe HPG/HPC GPUs from here `https://dgpu-docs.intel.com/`. Follow the driver installation instructions for your graphics card and operating system.

We tested Embree with the latest GPGPU driver Devel Release from 20220809. The Intel(R) Graphics Compute Runtime for oneAPI Level Zero and OpenCL(TM) Driver from that release is too old for Embree to work properly. Thus if no newer version of the GPGPU driver is available, you need to additionally install the latest compute runtime from here 22.43.24595.

Unfortunately, these compute runtime packages are only available for Ubuntu 22.04. You can also install a newer version of the compute runtime if available.

3.3 Windows

Embree is tested using the following compilers under Windows:

- Intel® oneAPI DPC++/C++ Compiler 2023.0.0
- oneAPI DPC++/C++ Compiler 2022-12-14
- Visual Studio 2019
- Visual Studio 2017
- Visual Studio 2015 (Update 1)
- Intel® Implicit SPMD Program Compiler 1.18.1
- Intel® Implicit SPMD Program Compiler 1.17.0
- Intel® Implicit SPMD Program Compiler 1.16.1
- Intel® Implicit SPMD Program Compiler 1.15.0
- Intel® Implicit SPMD Program Compiler 1.14.1
- Intel® Implicit SPMD Program Compiler 1.13.0
- Intel® Implicit SPMD Program Compiler 1.12.0

To compile Embree for AVX-512 you have to use the Intel® Compiler.

Embree supports using the Intel® Threading Building Blocks (TBB) as the tasking system. For performance and flexibility reasons we recommend using 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 `https://github.com/oneapi-src/oneTBB/releases` 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® Implicit SPMD Program Compiler (Intel® ISPC), which allows straightforward parallelization of an entire renderer. When installing Intel® ISPC, make sure to download an Intel® ISPC version from ispc.github.io that is compatible with your Visual Studio version. After installation, put the path to `ispc.exe` permanently into your `PATH` environment variable or you need to correctly set the `EMBREE_ISPC_EXECUTABLE` variable during CMake configuration to point to the ISPC executable. If you want to use Intel® ISPC, you have to enable `EMBREE_ISPC_SUPPORT` in CMake.

You additionally have to install CMake (version 3.1 or higher). Note that you need a native Windows CMake installation because CMake under Cygwin cannot generate solution files for Visual Studio.

3.3.1 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 compiler 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_v142".

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), because then some compiler-specific command line options cannot be 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.

The following CMake options are only available under Windows:

- `CMAKE_CONFIGURATION_TYPE`: List of generated configurations. The default value is Debug;Release;RelWithDebInfo.
- `USE_STATIC_RUNTIME`: Use the static version of the C/C++ runtime library. This option is turned OFF by default.

Use the generated Visual Studio solution file `embree4.sln` to compile the project.

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` extensions for the "Microsoft Visual C++" editor.

3.3.2 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 16 2019" ..
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
```

3.3.3 Building Embree - Using vcpkg

You can download and install Embree using the vcpkg dependency manager:

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg integrate install
./vcpkg install embree3
```

The Embree port in vcpkg is kept up to date by Microsoft team members and community contributors. If the version is out of date, please create an issue or pull request on the vcpkg repository.

3.4 Windows SYCL Compilation

There are two options to compile Embree with SYCL support: The open source "oneAPI DPC++ Compiler" or the "Intel(R) oneAPI DPC++/C++ Compiler". Other SYCL compilers are not supported. You will also need an installed version of Visual Studio that supports the C++17 standard, e.g. Visual Studio 2019.

The "oneAPI DPC++ Compiler" is more up-to-date than the "Intel(R) oneAPI DPC++/C++ Compiler" but less stable. The current tested version of the oneAPI DPC++ compiler is

- oneAPI DPC++ Compiler 2022-12-14

Download and unpack the archive and open the "x64 Native Tools Command Prompt" of Visual Studio and execute the following lines to properly configure the environment to use the oneAPI DPC++ compiler:

```
set "DPCPP_DIR=path_to_dpcpp_compiler"
set "PATH=%DPCPP_DIR%\bin;%PATH%"
set "PATH=%DPCPP_DIR%\lib;%PATH%"
set "CPATH=%DPCPP_DIR%\include;%CPATH%"
set "INCLUDE=%DPCPP_DIR%\include;%INCLUDE%"
set "LIB=%DPCPP_DIR%\lib;%LIB%"
```

The `path_to_dpcpp_compiler` should point to the unpacked oneAPI DPC++ compiler.

Now, you can configure Embree using CMake by executing the following command in the Embree root directory:

```
cmake -B build
      -G Ninja
      -D CMAKE_BUILD_TYPE=Release
      -D CMAKE_CXX_COMPILER=clang++
      -D CMAKE_C_COMPILER=clang
      -D EMBREE_SYCL_SUPPORT=ON
      -D TBB_ROOT=path_to_tbb\lib\cmake\tbb
```

This will create a directory `build` to use as the CMake build directory, and configure a release build that uses `clang++` and `clang` from the oneAPI DPC++ compiler.

The Ninja generator is currently the easiest way to use the oneAPI DPC++ compiler.

We also enable SYCL support in Embree using the `EMBREE_SYCL_SUPPORT` CMake option.

Alternatively, you can download and run the installer of the

- Intel(R) oneAPI DPC++/C++ Compiler 2023.0.0.

After installation, you can either open a regular `Command Prompt` and execute the `vars.bat` script in the `env` directory of the compiler install directory, for example

```
C:\Program Files (x86)\Intel\oneAPI\compiler\2023.0.0\env\vars.bat
```

or simply open the installed "Intel oneAPI command prompt for Intel 64 for Visual Studio".

Both ways will put the `icx` compiler executable from the Intel(R) oneAPI DPC++/C++ compiler in your path.

Now, you can configure Embree using CMake by executing the following command in the Embree root directory:

```
cmake -B build
      -G Ninja
      -D CMAKE_BUILD_TYPE=Release
      -D CMAKE_CXX_COMPILER=icx
      -D CMAKE_C_COMPILER=icx
      -D EMBREE_SYCL_SUPPORT=ON
      -D TBB_ROOT=path_to_tbb\lib\cmake\tbb
```

More information about setting up the Intel(R) oneAPI DPC++/C++ compiler can be found in the Development Reference Guide. Please note, that the Intel(R) oneAPI DPC++/C++ compiler requires at least CMake version 3.23 on Windows.

Independent of the DPC++ compiler choice, you can now build Embree using

```
cmake --build build
```

If you have problems with Ninja re-running CMake in an infinite loop, then first remove the "Re-run CMake if any of its inputs changed." section from the `build.ninja` file and run the above command again.

You can also create an Embree package using the following command:

```
cmake --build build --target package
```

Please see the Building Embree SYCL Applications section on how to build your application with such an Embree package.

3.4.1 Windows Graphics Driver Installation

In order to run the SYCL tutorials on HPG hardware, you first need to install the proper graphics drivers for your graphics card from `https://www.intel.com`. Embree will work with graphics driver version 101.4027 or later.

3.5 CMake Configuration

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

- `CMAKE_BUILD_TYPE`: Can be used to switch between Debug mode (Debug), Release mode (Release) (default), and Release mode with enabled assertions and debug symbols (RelWithDebInfo).
- `EMBREE_STACK_PROTECTOR`: Enables protection of return address from buffer overwrites. This option is OFF by default.

- EMBREE_ISPC_SUPPORT: Enables Intel® ISPC support of Embree. This option is OFF by default.
- EMBREE_SYCL_SUPPORT: Enables GPU support using SYCL. When this option is enabled you have to use some DPC++ compiler. Please see the sections Linux SYCL Compilation and Windows SYCL Compilation on supported DPC++ compilers. This option is OFF by default.
- EMBREE_SYCL_AOT_DEVICES: Selects a list of GPU devices for ahead-of-time (AOT) compilation of device code. Possible values are either, "none" which enables only just in time (JIT) compilation, or a list of the Embree-supported Xe GPUs for AOT compilation:
  - XE_HPG_CORE : Xe HPG devices
  - XE_HPC_CORE : Xe HPC devices

  One can also specify multiple devices separated by comma to compile ahead of time for multiple devices, e.g. "XE_HPG_CORE,XE_HP_CORE". When enabling AOT compilation for one or multiple devices, JIT compilation will always additionally be enabled in case the code is executed on a device no code is precompiled for.

  Execute "ocloc compile –help" for more details of possible devices to pass. Embree is only supported on Xe HPG/HPC and newer devices.

  Per default, this option is set to "none" to enable JIT compilation. We recommend using JIT compilation as this enables the use of specialization constants to reduce code complexity.
- EMBREE_STATIC_LIB: Builds Embree as a static library (OFF by default). Further multiple static libraries are generated for the different ISAs selected (e.g. `embree4.a`, `embree4_sse42.a`, `embree4_avx.a`, `embree4_avx2.a`, `embree4_avx512.a`). You have to link these libraries in exactly this order of increasing ISA.
- EMBREE_API_NAMESPACE: Specifies a namespace name to put all Embree API symbols inside. By default, no namespace is used and plain C symbols are exported.
- EMBREE_LIBRARY_NAME: Specifies the name of the Embree library file created. By default, the name embree4 is used.
- EMBREE_IGNORE_CMAKE_CXX_FLAGS: When enabled, Embree ignores default CMAKE_CXX_FLAGS. This option is turned ON by default.
- EMBREE_TUTORIALS: Enables build of Embree tutorials (default ON).
- EMBREE_BACKFACE_CULLING: Enables backface culling, i.e. only surfaces facing a ray can be hit. This option is turned OFF by default.
- EMBREE_COMPACT_POLYS: Enables compact tris/quads, i.e. only geomIDs and primIDs are stored inside the leaf nodes.
- EMBREE_FILTER_FUNCTION: Enables the intersection filter function feature (ON by default).
- EMBREE_RAY_MASK: Enables the ray masking feature (OFF by default).
- EMBREE_RAY_PACKETS: Enables ray packet traversal kernels. This feature is turned ON by default. When turned on packet traversal is used internally and packets passed to rtcIntersect4/8/16 are kept intact in callbacks (when the ISA of appropriate width is enabled).

- 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). This option is turned OFF by default.
- EMBREE_TASKING_SYSTEM: Chooses between Intel® Threading TBB Building Blocks (TBB), Parallel Patterns Library (PPL) (Windows only), or an internal tasking system (INTERNAL). By default, TBB is used.
- EMBREE_TBB_ROOT: If Intel® Threading Building Blocks (TBB) is used as a tasking system, search the library in this directory tree.
- EMBREE_TBB_COMPONENT: The component/library name of Intel® Threading Building Blocks (TBB). Embree searches for this library name (default: tbb) when TBB is used as the tasking system.
- EMBREE_TBB_POSTFIX: If Intel® Threading Building Blocks (TBB) is used as a tasking system, link to tbb.(so,dll,lib). Defaults to the empty string.
- EMBREE_TBB_DEBUG_ROOT: If Intel® Threading Building Blocks (TBB) is used as a tasking system, search the library in this directory tree in Debug mode. Defaults to EMBREE_TBB_ROOT.
- EMBREE_TBB_DEBUG_POSTFIX: If Intel® Threading Building Blocks (TBB) is used as a tasking system, link to tbb.(so,dll,lib) in Debug mode. Defaults to "_debug".
- EMBREE_MAX_ISA: Select highest supported ISA (SSE2, SSE4.2, AVX, AVX2, AVX512, or NONE). When set to NONE the EMBREE_ISA_* variables can be used to enable ISAs individually. By default, the option is set to AVX2.
- EMBREE_ISA_SSE2: Enables SSE2 when EMBREE_MAX_ISA is set to NONE. By default, this option is turned OFF.
- EMBREE_ISA_SSE42: Enables SSE4.2 when EMBREE_MAX_ISA is set to NONE. By default, this option is turned OFF.
- EMBREE_ISA_AVX: Enables AVX when EMBREE_MAX_ISA is set to NONE. By default, this option is turned OFF.
- EMBREE_ISA_AVX2: Enables AVX2 when EMBREE_MAX_ISA is set to NONE. By default, this option is turned OFF.
- EMBREE_ISA_AVX512: Enables AVX-512 for Skylake when EMBREE_MAX_ISA is set to NONE. By default, this option is turned OFF.
- EMBREE_GEOMETRY_TRIANGLE: Enables support for triangle geometries (ON by default).
- EMBREE_GEOMETRY_QUAD: Enables support for quad geometries (ON by default).
- EMBREE_GEOMETRY_CURVE: Enables support for curve geometries (ON by default).
- EMBREE_GEOMETRY_SUBDIVISION: Enables support for subdivision geometries (ON by default).
- EMBREE_GEOMETRY_INSTANCE: Enables support for instances (ON by default).
- EMBREE_GEOMETRY_USER: Enables support for user-defined geometries (ON by default).

- EMBREE_GEOMETRY_POINT: Enables support for point geometries (ON by default).
- EMBREE_CURVE_SELF_INTERSECTION_AVOIDANCE_FACTOR: Specifies a factor that controls the self-intersection avoidance feature for flat curves. Flat curve intersections which are closer than curve_radius*EMBREE_CURVE_SELF_INTERSECTION_AVOIDANCE_FACTOR to the ray origin are ignored. A value of 0.0f disables self-intersection avoidance while 2.0f is the default value.
- EMBREE_DISC_POINT_SELF_INTERSECTION_AVOIDANCE: Enables self-intersection avoidance for RTC_GEOMETRY_TYPE_DISC_POINT geometry type (ON by default). When enabled intersections are skipped if the ray origin lies inside the sphere defined by the point primitive.
- EMBREE_MIN_WIDTH: Enabled the min-width feature, which allows increasing the radius of curves and points to match some amount of pixels. See rtcSetGeometryMaxRadiusScale for more details.
- EMBREE_MAX_INSTANCE_LEVEL_COUNT: Specifies the maximum number of nested instance levels. Should be greater than 0; the default value is 1. Instances nested any deeper than this value will silently disappear in release mode, and cause assertions in debug mode.

Chapter 4
Embree API

The Embree API is a low-level C99 ray tracing API which can be used to build spatial index structures for 3D scenes and perform ray queries of different types.

The API can get used on the CPU using standard C, C++, and ISPC code and Intel GPUs by using SYCL code.

The Intel® Implicit SPMD Program Compiler (Intel® ISPC) version of the API, is almost identical to the standard C99 version, but contains additional functions that operate on ray packets with a size of the native SIMD width used by Intel® ISPC.

The SYCL version of the API is also mostly identical to the C99 version of the API, with some exceptions listed in section Embree SYCL API.

For simplicity this document refers to the C99 version of the API functions. For changes when upgrading from the Embree 3 to the current Embree 4 API see Section Upgrading from Embree 3 to Embree 4.

All API calls carry the prefix `rtc` (or `RTC` for types) which stands for **r**ay **t**racing **c**ore. The API supports scenes consisting of different geometry types such as triangle meshes, quad meshes (triangle pairs), grid meshes, flat curves, round curves, oriented curves, subdivision meshes, instances, and user-defined geometries. See Section Scene Object for more information.

Finding the closest hit of a ray segment with the scene (`rtcIntersect`-type functions), and determining whether any hit between a ray segment and the scene exists (`rtcOccluded`-type functions) are both supported. The API supports queries for single rays and ray packets. See Section Ray Queries for more information.

The API is designed in an object-oriented manner, e.g. it contains device objects (`RTCDevice` type), scene objects (`RTCScene` type), geometry objects (`RTCGeometry` type), buffer objects (`RTCBuffer` type), and BVH objects (`RTCBVH` type). All objects are reference counted, and handles can be released by calling the appropriate release function (e.g. `rtcReleaseDevice`) or retained by incrementing the reference count (e.g. `rtcRetainDevice`). In general, API calls that access the same object are not thread-safe, unless specified otherwise. However, attaching geometries to the same scene and performing ray queries in a scene is thread-safe.

4.1 Device Object

Embree supports a device concept, which allows different components of the application to use the Embree API without interfering with each other. An application typically first creates a device using the rtcNewDevice function (or rtcNewSYCLDevice when using SYCL for the GPU). This device can then be used to construct further objects, such as scenes and geometries. Before the application exits, it should release all devices by invoking rtcReleaseDevice. An application

typically creates only a single device. If required differently, it should only use a small number of devices at any given time.

Each user thread has its own error flag per device. If an error occurs when invoking an API function, this flag is set to an error code (if it isn't already set by a previous error). See Section rtcGetDeviceError for information on how to read the error code and Section rtcSetDeviceErrorFunction on how to register a callback that is invoked for each error encountered. It is recommended to always set a error callback function, to detect all errors.

4.2 Scene Object

A scene is a container for a set of geometries, and contains a spatial acceleration structure which can be used to perform different types of ray queries.

A scene is created using the `rtcNewScene` function call, and released using the `rtcReleaseScene` function call. To populate a scene with geometries use the `rtcAttachGeometry` call, and to detach them use the `rtcDetachGeometry` call. Once all scene geometries are attached, an `rtcCommitScene` call (or `rtcJoinCommitScene` call) will finish the scene description and trigger building of internal data structures. After the scene got committed, it is safe to perform ray queries (see Section Ray Queries) or to query the scene bounding box (see rtcGetSceneBounds and rtcGetSceneLinearBounds).

If scene geometries get modified or attached or detached, the `rtcCommitScene` call must be invoked before performing any further ray queries for the scene; otherwise the effect of the ray query is undefined. The modification of a geometry, committing the scene, and tracing of rays must always happen sequentially, and never at the same time. Any API call that sets a property of the scene or geometries contained in the scene count as scene modification, e.g. including setting of intersection filter functions.

Scene flags can be used to configure a scene to use less memory (`RTC_SCENE_FLAG_COMPACT`), use more robust traversal algorithms (`RTC_SCENE_FLAG_ROBUST`), and to optimize for dynamic content. See Section rtcSetSceneFlags for more details.

A build quality can be specified for a scene to balance between acceleration structure build performance and ray query performance. See Section rtcSetSceneBuildQuality for more details on build quality.

4.3 Geometry Object

A new geometry is created using the `rtcNewGeometry` function. Depending on the geometry type, different buffers must be bound (e.g. using `rtcSetSharedGeometryBuffer`) to set up the geometry data. In most cases, binding of a vertex and index buffer is required. The number of primitives and vertices of that geometry is typically inferred from the size of these bound buffers.

Changes to the geometry always must be committed using the `rtcCommitGeometry` call before using the geometry. After committing, a geometry is not included in any scene. A geometry can be added to a scene by using the `rtcAttachGeometry` function (to automatically assign a geometry ID) or using the `rtcAttachGeometryById` function (to specify the geometry ID manually). A geometry can get attached to multiple scenes.

All geometry types support multi-segment motion blur with an arbitrary number of equidistant time steps (in the range of 2 to 129) inside a user specified time range. Each geometry can have a different number of time steps and a different time range. The motion blur geometry is defined by linearly interpolating the geometries of neighboring time steps. To construct a motion blur geometry, first the number of time steps of the geometry must be specified using

the rtcSetGeometryTimeStepCount function, and then a vertex buffer for each time step must be bound, e.g. using the rtcSetSharedGeometryBuffer function. Optionally, a time range defining the start (and end time) of the first (and last) time step can be set using the rtcSetGeometryTimeRange function. This feature will also allow geometries to appear and disappear during the camera shutter time if the time range is a sub range of [0,1].

4.4 Ray Queries

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

Supported are single ray queries (rtcIntersect1 and rtcOccluded1) as well as ray packet queries for ray packets of size 4 (rtcIntersect4 and rtcOccluded4), ray packets of size 8 (rtcIntersect8 and rtcOccluded8), and ray packets of size 16 (rtcIntersect16 and rtcOccluded16).

See Sections rtcIntersect1 and rtcOccluded1 for a detailed description of how to set up and trace a ray.

See tutorial Triangle Geometry for a complete example of how to trace single rays and ray packets.

4.5 Point Queries

The API supports traversal of the BVH using a point query object that specifies a location and a query radius. For all primitives intersecting the according domain, a user defined callback function is called which allows queries such as finding the closest point on the surface geometries of the scene (see Tutorial Closest Point) or nearest neighbour queries (see Tutorial Voronoi).

See Section rtcPointQuery for a detailed description of how to set up point queries.

4.6 Collision Detection

The Embree API also supports collision detection queries between two scenes consisting only of user geometries. Embree only performs broadphase collision detection, the narrow phase detection can be performed through a callback function.

See Section rtcCollide for a detailed description of how to set up collision detection.

Seen tutorial Collision Detection for a complete example of collision detection being used on a simple cloth solver.

4.7 Filter Functions

The API supports filter functions that are invoked for each intersection found during the rtcIntersect-type or rtcOccluded-type calls.

The filter functions can be set per-geometry using the rtcSetGeometryIntersectFilterFunction and rtcSetGeometryOccludedFilterFunction calls. The former ones are called geometry intersection filter functions, the latter ones geometry occlusion filter functions. These filter functions are designed to be used to ignore intersections outside of a user-defined silhouette of a primitive, e.g. to model tree leaves using transparency textures.

The filter function can also get passed as arguments directly to the traversal functions, see section rtcInitIntersectArguments and rtcInitOccludedArguments for more details. These argument filter functions are designed to change the semantics of the ray query, e.g. to accumulate opacity for transparent shadows, count the number of surfaces along a ray, collect all hits along a ray, etc. The argument filter function must be enabled to be used for a scene using the `RTC_SCENE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS` scene flag. The callback is only invoked for geometries that enable the callback using the `rtcSetGeometryEnableFilterFunctionFromArguments` call, or enabled for all geometries when the `RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER` ray query flag is set.

4.8 BVH Build API

The internal algorithms to build a BVH are exposed through the `RTCBVH` object and `rtcBuildBVH` call. This call makes it possible to build a BVH in a user-specified format over user-specified primitives. See the documentation of the `rtcBuildBVH` call for more details.

Chapter 5
Embree SYCL API

Embree supports ray tracing on Intel GPUs by using the SYCL programming language. SYCL is a Khronos standardized C++ based language for single source heterogenous programming for acceleration offload, see the SYCL webpage for details.

The Embree SYCL API is designed for photorealistic rendering use cases, where scene setup is performed on the host, and rendering on the device. The Embree SYCL API is very similar to the standard Embree C99 API, and supports most of its features, such as all triangle-type geometries, all curve types and basis functions, point geometry types, user geometries, filter callbacks, multi-level instancing, and motion blur.

To enable SYCL support you have to include the `sycl.hpp` file before the Embree API headers:

```
#include <sycl/sycl.hpp>
#include <embree4/rtcore.h>
```

Next you need to initializes an Embree SYCL device using the `rtcNewSYCLDevice` API function by providing a SYCL context.

Embree provides the `rtcIsSYCLDeviceSupported` API function to check if some SYCL device is supported by Embree. You can also use the `rtcSYCLDeviceSelector` to conveniently select the first SYCL device that is supported by Embree, e.g.:

```
sycl::device device(rtcSYCLDeviceSelector);
sycl::queue queue(device, exception_handler);
sycl::context context(device);
RTCDevice device = rtcNewSYCLDevice(context,"");
```

Scenes created with an Embree SYCL device can only get used to trace rays using SYCL on the GPU, it is not possible to trace rays on the CPU with such a device. To render on the CPU and GPU in parallel, the user has to create a second Embree device and create a second scene to be used on the CPU.

Files containing SYCL code, have to get compiled with the Intel® oneAPI DPC++ compiler. Please see section Linux SYCL Compilation and Windows SYCL Compilation for supported compilers. The DPC++ compiler performs a two-phase compilation, where host code is compiled in a first phase, and device code compiled in a second compilation phase.

Standard Embree API functions for scene construction can get used on the host but not the device. Data buffers that are shared with Embree (e.g. for vertex of index buffers) have to get allocated as SYCL unified shared memory (USM memory), using the `sycl::malloc` or `sycl::aligned_alloc` calls with `sycl::usm::alloc::shared` property, or the sycl::aligned_alloc_shared call, e.g:

```
void* ptr = sycl::aligned_alloc(16, bytes, queue, sycl::usm::alloc::shared);
```

These shared allocations have to be valid during rendering, as Embree may access contained data when tracing rays. Embree does not support device-only memory allocations, as the BVH builder implemented on the CPU relies on reading the data buffers.

Device side rendering can get invoked by submitting a SYCL `parallel_for` to the SYCL queue:

```
const sycl::specialization_id<RTCFeatureFlags> feature_mask;

RTCFeatureFlags required_features = RTC_FEATURE_FLAG_TRIANGLE;

queue.submit([=](sycl::handler& cgh)
{
  cgh.set_specialization_constant<feature_mask>(required_features);

  cgh.parallel_for(sycl::range<1>(1),[=](sycl::id<1> item, sycl::kernel_handler kh)
  {
    RTCIntersectArguments args;
    rtcInitIntersectArguments(&args);

    const RTCFeatureFlags features = kh.get_specialization_constant<feature_mask>();
    args.feature_mask = features;

    struct RTCRayHit rayhit;
    rayhit.ray.org_x = ox;
    rayhit.ray.org_y = oy;
    rayhit.ray.org_z = oz;
    rayhit.ray.dir_x = dx;
    rayhit.ray.dir_y = dy;
    rayhit.ray.dir_z = dz;
    rayhit.ray.tnear = 0;
    rayhit.ray.tfar = std::numeric_limits<float>::infinity();
    rayhit.ray.mask = -1;
    rayhit.ray.flags = 0;
    rayhit.hit.geomID = RTC_INVALID_GEOMETRY_ID;
    rayhit.hit.instID[0] = RTC_INVALID_GEOMETRY_ID;

    rtcIntersect1(scene, &rayhit, &args);

    result->geomID = rayhit.hit.geomID;
    result->primID = rayhit.hit.primID;
    result->tfar = rayhit.ray.tfar;
  });
});
queue.wait_and_throw();
```

This example passes a feature mask using a specialization contant to the `rtcIntersect1` function, which is recommended for GPU rendering. For best performance, this feature mask should get used to enable only features required by the application to render the scene, e.g. just triangles in this example.

Inside the SYCL `parallel_for` loop you can use rendering related functions, such as the `rtcIntersect1` and `rtcOccluded1` functions to trace rays, `rtcForwardIntersect1` and `rtcForwardOccluded1` to continue object traversal from inside a user geometry callback, and `rtcGetGeometryUserDataFromScene` to get the user data pointer of some geometry.

Have a look at the Minimal tutorial for a minimal SYCL example.

5.1 SYCL JIT caching

Compile times for just in time compilation (JIT compilation) can be large. To resolve this issue we recommend enabling persistent JIT compilation caching inside your application, by setting the `SYCL_CACHE_PERSISTENT` environment variable to `1`, and the `SYCL_CACHE_DIR` environment variable to some proper directory where the JIT cache should get stored. These environment variables have to get set before the SYCL device is created, e.g:

```
setenv("SYCL_CACHE_PERSISTENT","1",1);
setenv("SYCL_CACHE_DIR","cache_dir",1);

sycl::device device(rtcSYCLDeviceSelector);
...
```

5.2 SYCL Memory Pooling

Memory Pooling is a mechanism where small USM memory allocations are packed into larger allocation blocks. This mode is required when your application performs many small USM allocations, as otherwise only a small fraction of GPU memory is usable and data transfer performance will be low.

Memory pooling is supported for USM allocations that are read-only by the device. The following example allocated device read-only memory with memory pooling support:

```
sycl::aligned_alloc_shared(align, bytes, queue,
  sycl::ext::oneapi::property::usm::device_read_only());
```

5.3 Embree SYCL Limitations

Embree only supports Xe HPC and HPG GPUs as SYCL devices, thus in particular the CPU and other GPUs cannot get used as a SYCL device. To render on the CPU just use the standard C99 API without relying on SYCL.

The SYCL language spec puts some restrictions to device functions, such as disallowing: global variable access, malloc, invokation of virtual functions, function pointers, runtime type information, exceptions, recursion, etc. See Section `5.4. Language Restrictions for device functions` of the SYCL specification for more details.

Using Intel's oneAPI DPC++ compiler invoking an indirectly called function is allowed, but we do not recommend this for performance reasons.

Some features are not supported by the Embree SYCL API thus cannot get used on the GPU:

- The packet tracing functions `rtcIntersect4/8/16` and `rtcOccluded4/8/16` are not supported in SYCL device side code. Using these functions makes no sense for SYCL, as the programming model is implicitely executed in SIMT mode on the GPU anyway.
- Filter and user geometry callbacks stored inside the geometry objects are not supported on SYCL. Please use the alternative approach of passing the function pointer through the `RTCIntersectArguments` (or `RTCOccludedArguments`) structures to the tracing function, which enables inlining on the GPU.
- The `rtcInterpolate` function cannot get used on the the device. For most primitive types the vertex data interpolation is anyway a trivial operation,

and an API call just introduces overheads. On the CPU that overhead is acceptable, but on the GPU it is not. The `rtcInterpolate` function does not know the geometry type it is interpolating over, thus its implementation on the GPU would contain a large switch statement for all potential geometry types.

- Tracing rays using `rtcIntersect1` and `rtcOccluded1` functions from user geometry callbacks is not supported in SYCL. Please use the tail recursive `rtcForwardIntersect1` and `rtcForwardOccluded1` calls instead.
- Subdivision surfaces are not supported for Embree SYCL devices.
- Collision detection (`rtcCollide` API call) is not supported in SYCL device side code.
- Point queries (`rtcPointQuery` API call) are not supported in SYCL device side code.

5.4 Embree SYCL Known Issues

- The SYCL support of Embree is in beta phase. Current functionality, quality, and GPU performance may not reflect that of the final product.
- Currently only the following Intel® Arc™ GPUs are support:
  - Intel® Arc™ A770 Graphics
  - Intel® Arc™ A750 Graphics
  - Intel® Arc™ A770M Graphics
  - Intel® Arc™ A730M Graphics
  - Intel® Arc™ A550M Graphics
- Intel® Data Center GPU Max Series is currently not supported.
- Ahead of time compilation is currently not working properly and you will get this error during compilation:

  ```
  llvm-foreach: Floating point exception (core dumped)
  ```

- Compilation with build configuration “debug” is currently not working on Windows.

Chapter 6

Upgrading from Embree 3 to Embree 4

This section summarizes API changes between Embree 3 and Embree4. Most of these changes are motivated by GPU performance and having a consistent API that works properly for the CPU and GPU.

- The API include folder got renamed from embree3 to embree4, to be able to install Embree 3 and Embree 4 side by side, without having conflicts in API folder.
- The `RTCIntersectContext` is renamed to `RTCRayQueryContext` and the `RTCIntersectContextFlags` got renamed to `RTCRayQueryFlags`.
- There are some changes to the `rtcIntersect` and `rtcOccluded` functions. Most members of the old intersect context have been moved to some optional `RTCIntersectArguments` (and `RTCOccludedArguments`) structures, which also contains a pointer to the new ray query context. The argument structs fulfill the task of providing additional advanced arguments to the traversal functions. The ray query context can get used to pass additional data to callbacks, and to maintain an instID stack in case instancing is done manually inside user geometry callbacks. The arguments struct is not available inside callbacks. This change was in particular necessary for SYCL to allow inlining of function pointers provided to the traversal functions, and to reduce the amount of state passed to callbacks, which both improves GPU performance. Most applications can just drop passing the ray query context to port to Embree 4.
- The `rtcFilterIntersection` and `rtcFilterOcclusion` API calls that invoke both, the geometry and argument version of the filter callback, from a user geometry callback are no longer supported. Instead applications should use the `rtcInvokeIntersectFilterFromGeometry` and `rtcInvokeOccludedFilterFromGeometry` API calls that invoke just the geometry version of the filter function, and invoke the argument filter function manually if required.
- The filter function passed as arguments to `rtcIntersect` and `rtcOccluded` functions is only invoked for some geometry if enabled through `rtcSetGeometryEnableFilterFunctionFromArguments` for that geometry. Alternatively, argument filter functions can get enabled for all geometries using the `RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER` ray query flag.

- User geometry callbacks get a valid vector as input to identify valid and invalid rays. In Embree 3 the user geometry callback just had to update the ray hit members when an intersection was found and perform no operation otherwise. In Embree 4 the callback additionally has to return valid=-1 when a hit was found, and valid=0 when no hit was found. This allows Embree to properly pass the new hit distance to the ray tracing hardware only in the case a hit was found.
- Further ray masking is enabled by default now as required by most applications and the default ray mask for geometries got changed from 0xFFFFFFFF to 0x1.
- The stream tracing functions `rtcIntersect1M`, `rtcIntersect1Mp`, `rtcIntersectNM`, `rtcIntersectNp`, `rtcOccluded1M`, `rtcOccluded1Mp`, `rtcOccludedNM`, and `rtcOccludedNp` got removed as they were rarely used and did not provide relevant performance benefits. As alternative the application can just iterate over `rtcIntersect1` and potentially `rtcIntersect4/8/16` to get similar performance.

To use Embree through SYCL on the CPU and GPU additional changes are required:

- Embree 3 allows to use `rtcIntersect` recursively from a user geometry or intersection filter callback to continue a ray inside an instantiated object. In Embree 4 using `rtcIntersect` recursively is disallowed on the GPU but still supported on the CPU. To properly continue a ray inside an instantiated object use the new `rtcForwardIntersect1` and `rtcForwardOccluded1` functions.
- The geometry object of Embree 4 is a host side only object, thus accessing it during rendering from the GPU is not allowed. Thus all API functions that take an RTCGeometry object as argument cannot get used during rendering. Thus in particular the `rtcGetGeometryUserData(RTCGeometry)` call cannot get used, but there is an alternative function `rtcGetGeometryUserDataFromScene(RTCScene scene,uint geomID)` that should get used instead.
- The user geometry callback and filter callback functions should get passed through the intersection and occlusion argument structures to the `rtcIntersect1` and `rtcOccluded1` functions directly to allow inlining. The experimental geometry version of the callbacks is disabled in SYCL and should not get used.
- The feature flags should get used in SYCL to minimal GPU code for optimal performance.
- The `rtcInterpolate` function cannot get used on the device, and vertex data interpolation should get implemented by the application.
- Indirectly called functions must be declared with `RTC_SYCL_INDIRECTLY_CALLABLE` when used as filter or user geometry callbacks.

Chapter 7

Embree API Reference

7.1 rtcNewDevice

NAME

`rtcNewDevice - creates a new device`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCDevice rtcNewDevice(const char* config);
```

DESCRIPTION

This function creates a new device to be used for CPU ray tracing and returns a handle to this device. The device object is reference counted with an initial reference count of 1. The handle can be released using the `rtcReleaseDevice` API call.

The device object acts as a class factory for all other object types. All objects created from the device (like scenes, geometries, etc.) hold a reference to the device, thus the device will not be destroyed unless these objects are destroyed first.

Objects are only compatible if they belong to the same device, e.g it is not allowed to create a geometry in one device and attach it to a scene created with a different device.

A configuration string (`config` argument) can be passed to the device construction. This configuration string can be `NULL` to use the default configuration.

The following configuration is supported:

- `threads=[int]`: Specifies a number of build threads to use. A value of 0 enables all detected hardware threads. By default all hardware threads are used.
- `user_threads=[int]`: Sets the number of user threads that can be used to join and participate in a scene commit using `rtcJoinCommitScene`. The tasking system will only use threads-user_threads many worker threads, thus if the app wants to solely use its threads to commit scenes, just set threads equal to user_threads. This option only has effect with the Intel(R) Threading Building Blocks (TBB) tasking system.
- `set_affinity=[0/1]`: When enabled, build threads are affinitized to hardware threads. This option is disabled by default on standard CPUs, and enabled by default on Xeon Phi Processors.

- `start_threads=[0/1]`: When enabled, the build threads are started upfront. This can be useful for benchmarking to exclude thread creation time. This option is disabled by default.
- `isa=[sse2,sse4.2,avx,avx2,avx512]`: Use specified ISA. By default the ISA is selected automatically.
- `max_isa=[sse2,sse4.2,avx,avx2,avx512]`: Configures the automated ISA selection to use maximally the specified ISA.
- `hugepages=[0/1]`: Enables or disables usage of huge pages. Under Linux huge pages are used by default but under Windows and macOS they are disabled by default.
- `enable_selockmemoryprivilege=[0/1]`: When set to 1, this enables the `SeLockMemoryPrivilege` privilege with is required to use huge pages on Windows. This option has an effect only under Windows and is ignored on other platforms. See Section Huge Page Support for more details.
- `verbose=[0,1,2,3]`: Sets the verbosity of the output. When set to 0, no output is printed by Embree, when set to a higher level more output is printed. By default Embree does not print anything on the console.
- `frequency_level=[simd128,simd256,simd512]`: Specifies the frequency level the application want to run on, which can be either:

  a) simd128 to run at highest frequency
  b) simd256 to run at AVX2-heavy frequency level
  c) simd512 to run at heavy AVX512 frequency level. When some frequency level is specified, Embree will avoid doing optimizations that may reduce the frequency level below the level specified. E.g. if your app does not use AVX instructions setting "frequency_level=simd128" will cause some CPUs to run at highest frequency, which may result in higher application performance if you do much shading. If you application heavily uses AVX code, you should best set the frequency level to simd256. Per default Embree tries to avoid reducing the frequency of the CPU by setting the simd256 level only when the CPU has no significant down clocking.

Different configuration options should be separated by commas, e.g.:

```
rtcNewDevice("threads=1,isa=avx");
```

EXIT STATUS

On success returns a handle of the created device. On failure returns `NULL` as device and sets a per-thread error code that can be queried using `rtcGetDeviceError(NULL)`.

SEE ALSO

rtcRetainDevice, rtcReleaseDevice, rtcNewSYCLDevice

7.2 rtcNewSYCLDevice

NAME

`rtcNewSYCLDevice - creates a new device to be used with SYCL`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCDevice rtcNewSYCLDevice(sycl::context context, const char* config);
```

DESCRIPTION

This function creates a new device to be used with SYCL for GPU rendering and returns a handle to this device. The device object is reference counted with an initial reference count of 1. The handle can get released using the `rtcReleaseDevice` API call.

The passed SYCL context (`context` argument) is used to allocate GPU data, thus only devices contained inside this context can be used for rendering. By default the GPU data is allocated on the first GPU device of the context, but this behavior can get changed with the rtcSetDeviceSYCLDevice function.

The device object acts as a class factory for all other object types. All objects created from the device (like scenes, geometries, etc.) hold a reference to the device, thus the device will not be destroyed unless these objects are destroyed first.

Objects are only compatible if they belong to the same device, e.g it is not allowed to create a geometry in one device and attach it to a scene created with a different device.

For an overview of configurations that can get passed (`config` argument) please see the rtcNewDevice function description.

EXIT STATUS

On success returns a handle of the created device. On failure returns `NULL` as device and sets a per-thread error code that can be queried using `rtcGetDeviceError(NULL)`.

SEE ALSO

rtcRetainDevice, rtcReleaseDevice, rtcNewDevice

7.3 rtcIsSYCLDeviceSupported

NAME

`rtcIsSYCLDeviceSupported - checks if some SYCL device is supported by Embree`

SYNOPSIS

```
#include <embree4/rtcore.h>

bool rtcIsSYCLDeviceSupported(const sycl::device sycl_device);
```

DESCRIPTION

This function can be used to check if some SYCL device (`sycl_device` argument) is supported by Embree.

EXIT STATUS

The function returns true if the SYCL device is supported by Embree and false otherwise. On failure an error code is set that can get queried using `rtcGetDeviceError`.

SEE ALSO

rtcSYCLDeviceSelector

7.4 rtcSYCLDeviceSelector

NAME

```
rtcSYCLDeviceSelector - SYCL device selector function to select
  devices supported by Embree
```

SYNOPSIS

```
#include <embree4/rtcore.h>

int rtcSYCLDeviceSelector(const sycl::device sycl_device);
```

DESCRIPTION

This function checks if the passed SYCL device (`sycl_device` arguments) is supported by Embree or not. This function can be used directly to select some supported SYCL device by using it as SYCL device selector function. For instance, the following code sequence selects an Embree supported SYCL device and creates an Embree device from it:

```
sycl::device sycl_device(rtcSYCLDeviceSelector);
sycl::queue sycl_queue(sycl_device);
sycl::context(sycl_device);
RTCDevice device = rtcNewSYCLDevice(sycl_context,nullptr);
```

EXIT STATUS

The function returns -1 if the SYCL device is supported by Embree and 1 otherwise. On failure an error code is set that can get queried using `rtcGetDeviceError`.

SEE ALSO

rtcIsSYCLDeviceSupported

7.5 rtcSetDeviceSYCLDevice

NAME

```
rtcSetDeviceSYCLDevice - sets the SYCL device to be used for memory allocations
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetDeviceSYCLDevice(RTCDevice device, const sycl::device sycl_device);
```

DESCRIPTION

This function sets the SYCL device (`sycl_device` argument) to be used to allocate GPU memory when using the specified Embree device (`device` argument). This SYCL device must be one of the SYCL devices contained inside the SYCL context used to create the Embree device.

EXIT STATUS

On failure an error code is set that can get queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewSYCLDevice

7.6 rtcRetainDevice

NAME

`rtcRetainDevice - increments the device reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcRetainDevice(RTCDevice device);
```

DESCRIPTION

Device objects are reference counted. The `rtcRetainDevice` function increments the reference count of the passed device object (`device` argument). This function together with `rtcReleaseDevice` allows to use the internal reference counting in a C++ wrapper class to manage the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewDevice, rtcReleaseDevice

7.7 rtcReleaseDevice

NAME

`rtcReleaseDevice - decrements the device reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcReleaseDevice(RTCDevice device);
```

DESCRIPTION

Device objects are reference counted. The `rtcReleaseDevice` function decrements the reference count of the passed device object (`device` argument). When the reference count falls to 0, the device gets destroyed.

All objects created from the device (like scenes, geometries, etc.) hold a reference to the device, thus the device will not get destroyed unless these objects are destroyed first.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewDevice, rtcRetainDevice

7.8 rtcGetDeviceProperty

NAME

`rtcGetDeviceProperty - queries properties of the device`

SYNOPSIS

```
#include <embree4/rtcore.h>

ssize_t rtcGetDeviceProperty(
  RTCDevice device,
  enum RTCDeviceProperty prop
);
```

DESCRIPTION

The `rtcGetDeviceProperty` function can be used to query properties (`prop` argument) of a device object (`device` argument). The returned property is an integer of type `ssize_t`.

Possible properties to query are:

- `RTC_DEVICE_PROPERTY_VERSION`: Queries the combined version number (MAJOR.MINOR.PATCH) with two decimal digits per component. E.g. for Embree 2.8.3 the integer 208003 is returned.
- `RTC_DEVICE_PROPERTY_VERSION_MAJOR`: Queries the major version number of Embree.
- `RTC_DEVICE_PROPERTY_VERSION_MINOR`: Queries the minor version number of Embree.
- `RTC_DEVICE_PROPERTY_VERSION_PATCH`: Queries the patch version number of Embree.
- `RTC_DEVICE_PROPERTY_NATIVE_RAY4_SUPPORTED`: Queries whether the `rtcIntersect4` and `rtcOccluded4` functions preserve packet size and ray order when invoking callback functions. This is only the case if Embree is compiled with `EMBREE_RAY_PACKETS` and `SSE2` (or `SSE4.2`) enabled, and if the machine it is running on supports `SSE2` (or `SSE4.2`).
- `RTC_DEVICE_PROPERTY_NATIVE_RAY8_SUPPORTED`: Queries whether the `rtcIntersect8` and `rtcOccluded8` functions preserve packet size and ray order when invoking callback functions. This is only the case if Embree is compiled with `EMBREE_RAY_PACKETS` and `AVX` (or `AVX2`) enabled, and if the machine it is running on supports `AVX` (or `AVX2`).
- `RTC_DEVICE_PROPERTY_NATIVE_RAY16_SUPPORTED`: Queries whether the `rtcIntersect16` and `rtcOccluded16` functions preserve packet size and ray order when invoking callback functions. This is only the case if Embree is compiled with `EMBREE_RAY_PACKETS` and `AVX512` enabled, and if the machine it is running on supports `AVX512`.
- `RTC_DEVICE_PROPERTY_RAY_MASK_SUPPORTED`: Queries whether ray masks are supported. This is only the case if Embree is compiled with `EMBREE_RAY_MASK` enabled.
- `RTC_DEVICE_PROPERTY_BACKFACE_CULLING_ENABLED`: Queries whether back face culling is enabled. This is only the case if Embree is compiled with `EMBREE_BACKFACE_CULLING` enabled.

- RTC_DEVICE_PROPERTY_COMPACT_POLYS_ENABLED: Queries whether compact polys is enabled. This is only the case if Embree is compiled with EMBREE_COMPACT_POLYS enabled.
- RTC_DEVICE_PROPERTY_FILTER_FUNCTION_SUPPORTED: Queries whether filter functions are supported, which is the case if Embree is compiled with EMBREE_FILTER_FUNCTION enabled.
- RTC_DEVICE_PROPERTY_IGNORE_INVALID_RAYS_ENABLED: Queries whether invalid rays are ignored, which is the case if Embree is compiled with EMBREE_IGNORE_INVALID_RAYS enabled.
- RTC_DEVICE_PROPERTY_TRIANGLE_GEOMETRY_SUPPORTED: Queries whether triangles are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_TRIANGLE enabled.
- RTC_DEVICE_PROPERTY_QUAD_GEOMETRY_SUPPORTED: Queries whether quads are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_QUAD enabled.
- RTC_DEVICE_PROPERTY_SUBDIVISION_GEOMETRY_SUPPORTED: Queries whether subdivision meshes are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_SUBDIVISION enabled.
- RTC_DEVICE_PROPERTY_CURVE_GEOMETRY_SUPPORTED: Queries whether curves are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_CURVE enabled.
- RTC_DEVICE_PROPERTY_POINT_GEOMETRY_SUPPORTED: Queries whether points are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_POINT enabled.
- RTC_DEVICE_PROPERTY_USER_GEOMETRY_SUPPORTED: Queries whether user geometries are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_USER enabled.
- RTC_DEVICE_PROPERTY_TASKING_SYSTEM: Queries the tasking system Embree is compiled with. Possible return values are:

  0. internal tasking system
  1. Intel Threading Building Blocks (TBB)
  2. Parallel Patterns Library (PPL)

- RTC_DEVICE_PROPERTY_JOIN_COMMIT_SUPPORTED: Queries whether rtcJoinCommitScene is supported. This is not the case when Embree is compiled with PPL or older versions of TBB.
- RTC_DEVICE_PROPERTY_PARALLEL_COMMIT_SUPPORTED: Queries whether rtcCommitScene can get invoked from multiple TBB worker threads concurrently. This feature is only supported starting with TBB 2019 Update 9.

EXIT STATUS

On success returns the value of the queried property. For properties returning a boolean value, the return value 0 denotes false and 1 denotes true.

On failure zero is returned and an error code is set that can be queried using rtcGetDeviceError.

7.9 rtcGetDeviceError

NAME

`rtcGetDeviceError - returns the error code of the device`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCError rtcGetDeviceError(RTCDevice device);
```

DESCRIPTION

Each thread has its own error code per device. If an error occurs when calling an API function, this error code is set to the occurred error if it stores no previous error. The `rtcGetDeviceError` function reads and returns the currently stored error and clears the error code. This assures that the returned error code is always the first error occurred since the last invocation of `rtcGetDeviceError`.

Possible error codes returned by `rtcGetDeviceError` are:

- `RTC_ERROR_NONE`: No error occurred.
- `RTC_ERROR_UNKNOWN`: An unknown error has occurred.
- `RTC_ERROR_INVALID_ARGUMENT`: An invalid argument was specified.
- `RTC_ERROR_INVALID_OPERATION`: The operation is not allowed for the specified object.
- `RTC_ERROR_OUT_OF_MEMORY`: There is not enough memory left to complete the operation.
- `RTC_ERROR_UNSUPPORTED_CPU`: The CPU is not supported as it does not support the lowest ISA Embree is compiled for.
- `RTC_ERROR_CANCELLED`: The operation got canceled by a 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 `rtcGetDeviceError` function. For all other invocations of `rtcGetDeviceError`, a proper device pointer must be specified.

EXIT STATUS

Returns the error code for the device.

SEE ALSO

rtcSetDeviceErrorFunction

7.10 rtcSetDeviceErrorFunction

NAME

`rtcSetDeviceErrorFunction - sets an error callback function for the device`

SYNOPSIS

```
#include <embree4/rtcore.h>

typedef void (*RTCErrorFunction)(
  void* userPtr,
  RTCError code,
  const char* str
);

void rtcSetDeviceErrorFunction(
  RTCDevice device,
  RTCErrorFunction error,
  void* userPtr
);
```

DESCRIPTION

Using the `rtcSetDeviceErrorFunction` call, it is possible to set a callback function (`error` argument) with payload (`userPtr` argument), which is called whenever an error occurs for the specified device (`device` argument).

Only a single callback function can be registered per device, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

When the registered callback function is invoked, it gets passed the user-defined payload (`userPtr` argument as specified at registration time), the error code (`code` argument) of the occurred error, as well as a string (`str` argument) that further describes the error.

The error code is also set if an error callback function is registered.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetDeviceError

7.11 rtcSetDeviceMemoryMonitorFunction

NAME

```
rtcSetDeviceMemoryMonitorFunction - registers a callback function
  to track memory consumption
```

SYNOPSIS

```
#include <embree4/rtcore.h>

typedef bool (*RTCMemoryMonitorFunction)(
  void* userPtr,
  ssize_t bytes,
  bool post
);

void rtcSetDeviceMemoryMonitorFunction(
  RTCDevice device,
  RTCMemoryMonitorFunction memoryMonitor,
  void* userPtr
);
```

DESCRIPTION

Using the `rtcSetDeviceMemoryMonitorFunction` call, it is possible to register a callback function (`memoryMonitor` argument) with payload (`userPtr` argument) for a device (`device` argument), which is called whenever internal memory is allocated or deallocated by objects of that device. Using this 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.

Only a single callback function can be registered per device, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

Once registered, the Embree device will invoke the memory monitor callback function before or after it allocates or frees important memory blocks. The callback function gets passed the payload as specified at registration time (`userPtr` argument), the number of bytes allocated or deallocated (`bytes` argument), and whether the callback is invoked after the allocation or deallocation took place (`post` argument). 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 `bytes` input parameter provided to the callback function. This parameter will be >0 for allocations and <0 for deallocations.

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_ERROR_OUT_OF_MEMORY` error code. Issuing multiple cancel requests from different threads is allowed. Canceling will only happen when the callback was called for allocations (bytes > 0), otherwise the cancel request will be ignored.

If a callback to cancel was invoked before the allocation happens (`post == false`), then the `bytes` parameter should not be accumulated, as the allocation will never happen. If the callback to cancel was invoked after the allocation happened (`post == true`), then the `bytes` parameter should be accumulated, as the allocation properly happened and a deallocation will later free that data block.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewDevice

7.12 rtcNewScene

NAME

`rtcNewScene - creates a new scene`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCScene rtcNewScene(RTCDevice device);
```

DESCRIPTION

This function creates a new scene bound to the specified device (`device` argument), and returns a handle to this scene. The scene object is reference counted with an initial reference count of 1. The scene handle can be released using the `rtcReleaseScene` API call.

EXIT STATUS

On success a scene handle is returned. On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainScene, rtcReleaseScene

7.13 rtcGetSceneDevice

NAME

`rtcGetSceneDevice - returns the device the scene got created in`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCDevice rtcGetSceneDevice(RTCScene scene);
```

DESCRIPTION

This function returns the device object the scene got created in. The returned handle own one additional reference to the device object, thus you should need to call `rtcReleaseDevice` when the returned handle is no longer required.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcReleaseDevice

7.14 rtcRetainScene

NAME

`rtcRetainScene - increments the scene reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcRetainScene(RTCScene scene);
```

DESCRIPTION

Scene objects are reference counted. The `rtcRetainScene` function increments the reference count of the passed scene object (`scene` argument). This function together with `rtcReleaseScene` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewScene, rtcReleaseScene

7.15 rtcReleaseScene

NAME

`rtcReleaseScene - decrements the scene reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcReleaseScene(RTCScene scene);
```

DESCRIPTION

Scene objects are reference counted. The `rtcReleaseScene` function decrements the reference count of the passed scene object (`scene` argument). When the reference count falls to 0, the scene gets destroyed.

The scene holds a reference to all attached geometries, thus if the scene gets destroyed, all geometries get detached and their reference count decremented.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewScene, rtcRetainScene

7.16 rtcAttachGeometry

NAME

`rtcAttachGeometry - attaches a geometry to the scene`

SYNOPSIS

```
#include <embree4/rtcore.h>

unsigned int rtcAttachGeometry(
  RTCScene scene,
  RTCGeometry geometry
);
```

DESCRIPTION

The `rtcAttachGeometry` function attaches a geometry (`geometry` argument) to a scene (`scene` argument) and assigns a geometry ID to that geometry. All geometries attached to a scene are defined to be included inside the scene. A geometry can get attached to multiple scenes. The geometry ID is unique for the scene, and is used to identify the geometry when hit by a ray during ray queries.

This function is thread-safe, thus multiple threads can attach geometries to a scene in parallel.

The geometry IDs are assigned sequentially, starting from 0, as long as no geometry got detached. If geometries got detached, the implementation will reuse IDs 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. Alternatively, the application can also use per-geometry user data to map to its geometry representation. See `rtcSetGeometryUserData` and `rtcGetGeometryUserData` for more information.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryUserData, rtcGetGeometryUserData

7.17 rtcAttachGeometryByID

NAME

```
rtcAttachGeometryByID - attaches a geometry to the scene
  using a specified geometry ID
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcAttachGeometryByID(
  RTCScene scene,
  RTCGeometry geometry,
  unsigned int geomID
);
```

DESCRIPTION

The `rtcAttachGeometryByID` function attaches a geometry (`geometry` argument) to a scene (`scene` argument) and assigns a user provided geometry ID (`geomID` argument) to that geometry. All geometries attached to a scene are defined to be included inside the scene. A geometry can get attached to multiple scenes. The passed user-defined geometry ID is used to identify the geometry when hit by a ray during ray queries. Using this function, it is possible to share the same IDs to refer to geometries inside the application and Embree.

This function is thread-safe, thus multiple threads can attach geometries to a scene in parallel.

The user-provided geometry ID must be unused in the scene, otherwise the creation of the geometry will fail. Further, the user-provided geometry IDs should be compact, as Embree internally creates 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.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcAttachGeometry

7.18 rtcDetachGeometry

NAME

`rtcDetachGeometry - detaches a geometry from the scene`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcDetachGeometry(RTCScene scene, unsigned int geomID);
```

DESCRIPTION

This function detaches a geometry identified by its geometry ID (`geomID` argument) from a scene (`scene` argument). When detached, the geometry is no longer contained in the scene.

This function is thread-safe, thus multiple threads can detach geometries from a scene at the same time.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcAttachGeometry, rtcAttachGeometryByID

7.19 rtcGetGeometry

NAME

```
rtcGetGeometry - returns the geometry bound to
  the specified geometry ID
```

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry rtcGetGeometry(RTCScene scene, unsigned int geomID);
```

DESCRIPTION

The `rtcGetGeometry` function returns the geometry that is bound to the specified geometry ID (`geomID` argument) for the specified scene (`scene` argument). This function just looks up the handle and does *not* increment the reference count. If you want to get ownership of the handle, you need to additionally call `rtcRetainGeometry`.

This function is not thread safe and thus can be used during rendering. However, it is generally recommended to store the geometry handle inside the application's geometry representation and look up the geometry handle from that representation directly.

If you need a thread safe version of this function please use rtcGetGeometryThreadSafe.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcAttachGeometry, rtcAttachGeometryByID, rtcGetGeometryThreadSafe

7.20 rtcGetGeometryThreadSafe

NAME

`rtcGetGeometryThreadSafe - returns the geometry bound to the specified geometry ID`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry rtcGetGeometryThreadSafe(RTCScene scene, unsigned int geomID);
```

DESCRIPTION

The `rtcGetGeometryThreadSafe` function returns the geometry that is bound to the specified geometry ID (`geomID` argument) for the specified scene (`scene` argument). This function just looks up the handle and does *not* increment the reference count. If you want to get ownership of the handle, you need to additionally call `rtcRetainGeometry`.

This function is thread safe and should NOT get used during rendering. If you need a fast non-thread safe version during rendering please use the rtcGetGeometry function.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcAttachGeometry, rtcAttachGeometryByID, rtcGetGeometry

7.21 rtcCommitScene

NAME

`rtcCommitScene - commits scene changes`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcCommitScene(RTCScene scene);
```

DESCRIPTION

The `rtcCommitScene` function commits all changes for the specified scene (`scene` argument). This internally triggers building of a spatial acceleration structure for the scene using all available worker threads. Ray queries can be performed only after committing all scene changes.

If the application uses TBB 2019 Update 9 or later for parallelization of rendering, lazy scene construction during rendering is supported by `rtcCommitScene`. Therefore `rtcCommitScene` can get called from multiple TBB worker threads concurrently for the same scene. The `rtcCommitScene` function will then internally isolate the scene construction using a tbb::isolated_task_group. The alternative approach of using `rtcJoinCommitScene` which uses an tbb:task_arena internally, is not recommended due to it's high runtime overhead.

If scene geometries get modified or attached or detached, the `rtcCommitScene` call must be invoked before performing any further ray queries for the scene; otherwise the effect of the ray query is undefined. The modification of a geometry, committing the scene, and tracing of rays must always happen sequentially, and never at the same time. Any API call that sets a property of the scene or geometries contained in the scene count as scene modification, e.g. including setting of intersection filter functions.

The kind of acceleration structure built can be influenced using scene flags (see `rtcSetSceneFlags`), and the quality can be specified using the `rtcSetSceneBuildQuality` function.

Embree silently ignores primitives during spatial acceleration structure construction that would cause numerical issues, e.g. primitives containing NaNs, INFs, or values greater than 1.844E18f (as no reasonable calculations can be performed with such values without causing overflows).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcJoinCommitScene

7.22 rtcJoinCommitScene

NAME

`rtcJoinCommitScene - commits the scene from multiple threads`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcJoinCommitScene(RTCScene scene);
```

DESCRIPTION

The `rtcJoinCommitScene` function commits all changes for the specified scene (`scene` argument). The scene commit internally triggers building of a spatial acceleration structure for the scene. Ray queries can be performed after scene changes got properly committed.

The `rtcJoinCommitScene` function can get called from multiple user threads which will all cooperate in the build operation. All threads calling into this function will return from `rtcJoinCommitScene` after the scene commit is finished. All threads must consistently call `rtcJoinCommitScene` and not `rtcCommitScene`.

In contrast to the `rtcCommitScene` function, the `rtcJoinCommitScene` function can be called from multiple user threads, while the `rtcCommitScene` can only get called from multiple TBB worker threads when used concurrently. For optimal performance we strongly recommend using TBB inside the application together with the `rtcCommitScene` function and to avoid using the `rtcJoinCommitScene` function.

The `rtcJoinCommitScene` 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 `rtcJoinCommitScene` on that just generated scene. During construction, further threads reaching the not-yet-built scene can join the build operation by also invoking `rtcJoinCommitScene`. A thread that calls `rtcJoinCommitScene` after the build finishes will directly return from the `rtcJoinCommitScene` call.

Multiple scene commit operations on different scenes can be running at the same time, hence it is possible to commit many small scenes in parallel, distributing the commits to many threads.

When using Embree with the Intel® Threading Building Blocks (which is the default), threads that call `rtcJoinCommitScene` 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 `rtcJoinCommitScene` to join a running build will just trigger the build and wait for the build to finish. Further, old TBB versions with `TBB_INTERFACE_VERSION_MAJOR < 8` do not support `rtcJoinCommitScene`, and invoking this function will result in an error.

When using Embree with the internal tasking system, only threads that call `rtcJoinCommitScene` will perform the build operation, and no additional worker threads will be scheduled.

When using Embree with the Parallel Patterns Library (PPL), `rtcJoinCommitScene` is not supported and calling that function will result in an error.

To detect whether `rtcJoinCommitScene` is supported, use the `rtcGetDeviceProperty` function.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcCommitScene, rtcGetDeviceProperty

7.23 rtcSetSceneProgressMonitorFunction

NAME

```
rtcSetSceneProgressMonitorFunction - registers a callback
  to track build progress
```

SYNOPSIS

```
#include <embree4/rtcore.h>

typedef bool (*RTCProgressMonitorFunction)(
  void* ptr,
  double n
);

void rtcSetSceneProgressMonitorFunction(
  RTCScene scene,
  RTCProgressMonitorFunction progress,
  void* userPtr
);
```

DESCRIPTION

Embree supports a progress monitor callback mechanism that can be used to report progress of hierarchy build operations and to cancel build operations.

The `rtcSetSceneProgressMonitorFunction` registers a progress monitor callback function (`progress` argument) with payload (`userPtr` argument) for the specified scene (`scene` argument).

Only a single callback function can be registered per scene, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

Once registered, Embree will invoke the callback function multiple times during hierarchy build operations of the scene, by passing the payload as set at registration time (`userPtr` argument), and a double in the range $[0, 1]$ which estimates the progress of the operation (`n` argument). The callback function might be 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_ERROR_CANCELLED` error code. Issuing multiple cancel requests for the same build operation is allowed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewScene

7.24 rtcSetSceneBuildQuality

NAME

```
rtcSetSceneBuildQuality - sets the build quality for
  the scene
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetSceneBuildQuality(
  RTCScene scene,
  enum RTCBuildQuality quality
);
```

DESCRIPTION

The `rtcSetSceneBuildQuality` function sets the build quality (`quality` argument) for the specified scene (`scene` argument). Possible values for the build quality are:

- `RTC_BUILD_QUALITY_LOW`: Create lower quality data structures, e.g. for dynamic scenes. A two-level spatial index structure is built when enabling this mode, which supports fast partial scene updates, and allows for setting a per-geometry build quality through the `rtcSetGeometryBuildQuality` function.
- `RTC_BUILD_QUALITY_MEDIUM`: Default build quality for most usages. Gives a good compromise between build and render performance.
- `RTC_BUILD_QUALITY_HIGH`: Create higher quality data structures for final-frame rendering. For certain geometry types this enables a spatial split BVH. When high quality mode is enabled, filter callbacks may be invoked multiple times for the same geometry.

Selecting a higher build quality results in better rendering performance but slower scene commit times. The default build quality for a scene is `RTC_BUILD_QUALITY_MEDIUM`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuildQuality

7.25 rtcSetSceneFlags

NAME

`rtcSetSceneFlags - sets the flags for the scene`

SYNOPSIS

```
#include <embree4/rtcore.h>

enum RTCSceneFlags
{
  RTC_SCENE_FLAG_NONE                     = 0,
  RTC_SCENE_FLAG_DYNAMIC                  = (1 << 0),
  RTC_SCENE_FLAG_COMPACT                  = (1 << 1),
  RTC_SCENE_FLAG_ROBUST                   = (1 << 2),
  RTC_SCENE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS = (1 << 3)
};

void rtcSetSceneFlags(RTCScene scene, enum RTCSceneFlags flags);
```

DESCRIPTION

The `rtcSetSceneFlags` function sets the scene flags (`flags` argument) for the specified scene (`scene` argument). Possible scene flags are:

- `RTC_SCENE_FLAG_NONE`: No flags set.
- `RTC_SCENE_FLAG_DYNAMIC`: Provides better build performance for dynamic scenes (but also higher memory consumption).
- `RTC_SCENE_FLAG_COMPACT`: Uses compact acceleration structures and avoids algorithms that consume much memory.
- `RTC_SCENE_FLAG_ROBUST`: Uses acceleration structures that allow for robust traversal, and avoids optimizations that reduce arithmetic accuracy. This mode is typically used for avoiding artifacts caused by rays shooting through edges of neighboring primitives.
- `RTC_SCENE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS`: Enables scene support for filter functions passed as argument to the traversal functions. See Section rtcInitIntersectArguments and rtcInitOccludedArguments for more details.

Multiple flags can be enabled using an `or` operation, e.g. `RTC_SCENE_FLAG_COMPACT | RTC_SCENE_FLAG_ROBUST`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetSceneFlags

7.26 rtcGetSceneFlags

NAME

`rtcGetSceneFlags - returns the flags of the scene`

SYNOPSIS

```
#include <embree4/rtcore.h>

enum RTCSceneFlags rtcGetSceneFlags(RTCScene scene);
```

DESCRIPTION

Queries the flags of a scene. This function can be useful when setting individual flags, e.g. to just set the robust mode without changing other flags the following way:

```
RTCSceneFlags flags = rtcGetSceneFlags(scene);
rtcSetSceneFlags(scene, RTC_SCENE_FLAG_ROBUST | flags);
```

EXIT STATUS

On failure `RTC_SCENE_FLAG_NONE` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetSceneFlags

7.27 rtcGetSceneBounds

NAME

`rtcGetSceneBounds - returns the axis-aligned bounding box of the scene`

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCORE_ALIGN(16) RTCBounds
{
  float lower_x, lower_y, lower_z, align0;
  float upper_x, upper_y, upper_z, align1;
};

void rtcGetSceneBounds(
  RTCScene scene,
  struct RTCBounds* bounds_o
);
```

DESCRIPTION

The `rtcGetSceneBounds` function queries the axis-aligned bounding box of the specified scene (`scene` argument) and stores that bounding box to the provided destination pointer (`bounds_o` argument). The stored bounding box consists of lower and upper bounds for the x, y, and z dimensions as specified by the `RTCBounds` structure.

The provided destination pointer must be aligned to 16 bytes. The function may be invoked only after committing the scene; otherwise the result is undefined.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetSceneLinearBounds, rtcCommitScene, rtcJoinCommitScene

7.28 rtcGetSceneLinearBounds

NAME

`rtcGetSceneLinearBounds - returns the linear bounds of the scene`

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCORE_ALIGN(16) RTCLinearBounds
{
  RTCBounds bounds0;
  RTCBounds bounds1;
};

void rtcGetSceneLinearBounds(
  RTCScene scene,
  struct RTCLinearBounds* bounds_o
);
```

DESCRIPTION

The `rtcGetSceneLinearBounds` function queries the linear bounds of the specified scene (`scene` argument) and stores them to the provided destination pointer (`bounds_o` argument). The stored linear bounds consist of bounding boxes for time 0 (`bounds0` member) and time 1 (`bounds1` member) as specified by the `RTCLinearBounds` structure. Linearly interpolating these bounds to a specific time `t` yields bounds for the geometry at that time.

The provided destination pointer must be aligned to 16 bytes. The function may be called only after committing the scene, otherwise the result is undefined.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetSceneBounds, rtcCommitScene, rtcJoinCommitScene

7.29 rtcNewGeometry

NAME

`rtcNewGeometry - creates a new geometry object`

SYNOPSIS

```
#include <embree4/rtcore.h>

enum RTCGeometryType
{
 RTC_GEOMETRY_TYPE_TRIANGLE,
 RTC_GEOMETRY_TYPE_QUAD,
 RTC_GEOMETRY_TYPE_SUBDIVISION,
 RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE,
 RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE,
 RTC_GEOMETRY_TYPE_GRID,
 RTC_GEOMETRY_TYPE_SPHERE_POINT,
 RTC_GEOMETRY_TYPE_DISC_POINT,
 RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT,
 RTC_GEOMETRY_TYPE_USER,
 RTC_GEOMETRY_TYPE_INSTANCE
};

RTCGeometry rtcNewGeometry(
  RTCDevice device,
  enum RTCGeometryType type
);
```

DESCRIPTION

Geometries are objects that represent an array of primitives of the same type. The `rtcNewGeometry` function creates a new geometry of specified type (`type` argument) bound to the specified device (`device` argument) and returns a handle to this geometry. The geometry object is reference counted with an initial reference count of 1. The geometry handle can be released using the `rtcReleaseGeometry` API call.

Supported geometry types are triangle meshes (`RTC_GEOMETRY_TYPE_TRIANGLE` type), quad meshes (triangle pairs) (`RTC_GEOMETRY_TYPE_QUAD` type), Catmull-Clark subdivision surfaces (`RTC_GEOMETRY_TYPE_SUBDIVISION` type), curve geometries with different bases (`RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE`,

RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE, RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE, RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE, RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE, RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE, RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE, RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE, RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE, RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE types) grid meshes (RTC_GEOMETRY_TYPE_GRID), point geometries (RTC_GEOMETRY_TYPE_SPHERE_POINT, RTC_GEOMETRY_TYPE_DISC_POINT, RTC_TYPE_ORIENTED_DISC_POINT), user-defined geometries (RTC_GEOMETRY_TYPE_USER), and instances (RTC_GEOMETRY_TYPE_INSTANCE).

The types RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE, RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE, and RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE will treat the curve as a sweep surface of a varying-radius circle swept tangentially along the curve. The types RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE, RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE, and RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE use ray-facing ribbons as a faster-to-intersect approximation.

After construction, geometries are enabled by default and not attached to any scene. Geometries can be disabled (rtcDisableGeometry call), and enabled again (rtcEnableGeometry call). A geometry can be attached to multiple scenes using the rtcAttachGeometry call (or rtcAttachGeometryByID call), and detached using the rtcDetachGeometry call. During attachment, a geometry ID is assigned to the geometry (or assigned by the user when using the rtcAttachGeometryByID call), which uniquely identifies the geometry inside that scene. This identifier is returned when primitives of the geometry are hit in later ray queries for the scene.

Geometries can also be modified, including their vertex and index buffers. After modifying a buffer, rtcUpdateGeometryBuffer must be called to notify that the buffer got modified.

The application can use the rtcSetGeometryUserData function to set a user data pointer to its own geometry representation, and later read out this pointer using the rtcGetGeometryUserData function.

After setting up the geometry or modifying it, rtcCommitGeometry must be called to finish the geometry setup. After committing the geometry, vertex data interpolation can be performed using the rtcInterpolate and rtcInterpolateN functions.

A build quality can be specified for a geometry using the rtcSetGeometryBuildQuality function, to balance between acceleration structure build performance and ray query performance. The build quality per geometry will be used if a two-level acceleration structure is built internally, which is the case if the RTC_BUILD_QUALITY_LOW is set as the scene build quality. See Section rtcSetSceneBuildQuality for more details.

EXIT STATUS

On failure NULL is returned and an error code is set that can be queried using rtcGetDeviceError.

SEE ALSO

rtcEnableGeometry, rtcDisableGeometry, rtcAttachGeometry, rtcAttachGeometryByID, rtcUpdateGeometryBuffer, rtcSetGeometryUserData, rtcGetGeometryUserData, rtcCommitGeometry, rtcInterpolate, rtcInterpolateN, rtcSetGeometryBuildQuality, rtcSetSceneBuildQuality, RTC_GEOMETRY_TYPE_TRIANGLE,

RTC_GEOMETRY_TYPE_QUAD, RTC_GEOMETRY_TYPE_SUBDIVISION, RTC_GEOMETRY_TYPE_CURVE, RTC_GEOMETRY_TYPE_GRID, RTC_GEOMETRY_TYPE_POINT, RTC_GEOMETRY_TYPE_USER, RTC_GEOMETRY_TYPE_INSTANCE

7.30 RTC_GEOMETRY_TYPE_TRIANGLE

NAME

`RTC_GEOMETRY_TYPE_TRIANGLE - triangle geometry type`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_TRIANGLE);
```

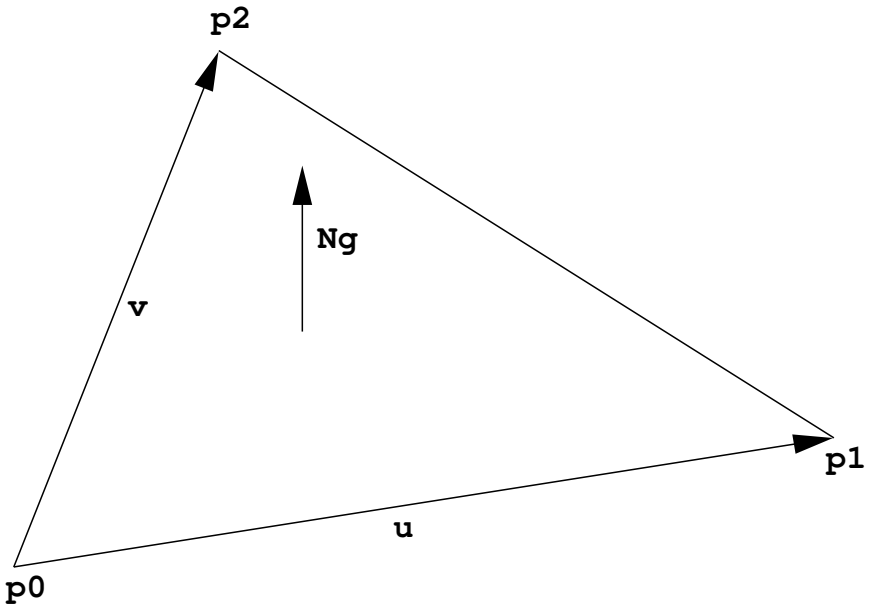
DESCRIPTION

Triangle meshes are created by passing `RTC_GEOMETRY_TYPE_TRIANGLE` to the `rtcNewGeometry` function call. The triangle indices can be specified by setting an index buffer (`RTC_BUFFER_TYPE_INDEX` type) and the triangle vertices by setting a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` type). See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers. The index buffer must contain an array of three 32-bit indices per triangle (`RTC_FORMAT_UINT3` format) and the number of primitives is inferred from the size of that buffer. The vertex buffer must contain an array of single precision `x`, `y`, `z` floating point coordinates (`RTC_FORMAT_FLOAT3` format), and the number of vertices are inferred from the size of that buffer. The vertex buffer can be at most 16 GB large.

The parametrization of a triangle uses the first vertex `p0` as base point, the vector `p1 - p0` as u-direction and the vector `p2 - p0` as v-direction. Thus vertex attributes `t0,t1,t2` can be linearly interpolated over the triangle the following way:

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

A triangle whose vertices are laid out counter-clockwise has its geometry normal pointing upwards outside the front face, like illustrated in the following picture:



For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for

each time step can be set using different buffer slots, and all these buffers have to have the same stride and size.

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

EXIT STATUS

On failure `NULL` is returned and an error code is set that be get queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

7.31 RTC_GEOMETRY_TYPE_QUAD

NAME

`RTC_GEOMETRY_TYPE_QUAD - quad geometry type`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_QUAD);
```

DESCRIPTION

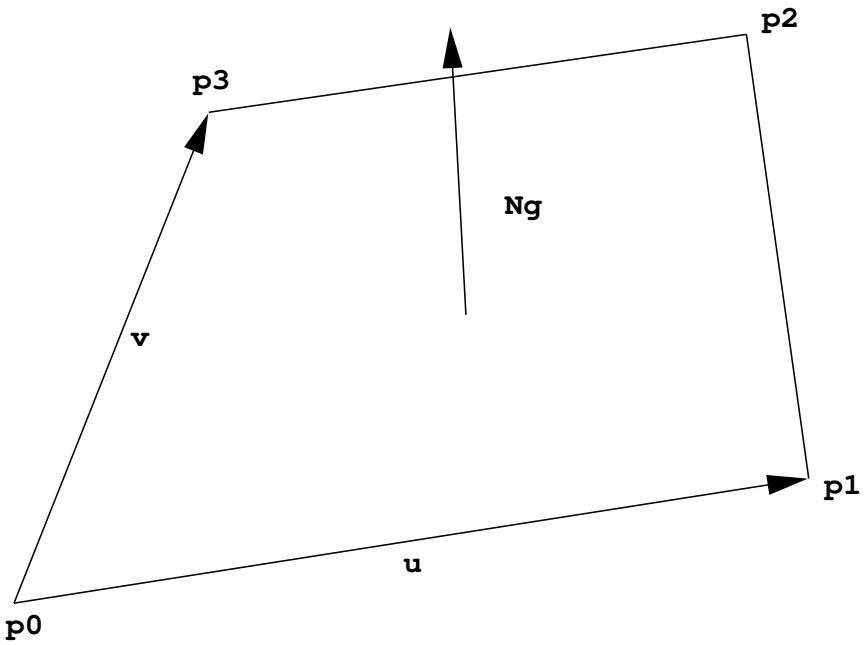
Quad meshes are created by passing `RTC_GEOMETRY_TYPE_QUAD` to the `rtcNewGeometry` function call. The quad indices can be specified by setting an index buffer (`RTC_BUFFER_TYPE_INDEX` type) and the quad vertices by setting a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` type). See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers. The index buffer contains an array of four 32-bit indices per quad (`RTC_FORMAT_UINT4` format), and the number of primitives is inferred from the size of that buffer. The vertex buffer contains an array of single precision `x`, `y`, `z` floating point coordinates (`RTC_FORMAT_FLOAT3` format), and the number of vertices is inferred from the size of that buffer. The vertex buffer can be at most 16 GB large.

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` are in the range 0 to 1. Thus the parametrization of a quad uses the first vertex `p0` as base point, and the vector `p1 - p0` as u-direction, and `p3 - p0` as v-direction. Thus vertex attributes `t0,t1,t2,t3` can be bilinearly interpolated over the quadrilateral the following way:

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

Mixed triangle/quad meshes are supported by encoding a triangle as a quad, which can be achieved by replicating the last triangle vertex (`v0,v1,v2 -> v0,v1,v2,v2`). This way the second triangle is a line (which can never get hit), and the parametrization of the first triangle is compatible with the standard triangle parametrization.

A quad whose vertices are laid out counter-clockwise has its geometry normal pointing upwards outside the front face, like illustrated in the following picture.



For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

7.32 RTC_GEOMETRY_TYPE_GRID

NAME

RTC_GEOMETRY_TYPE_GRID - grid geometry type

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_GRID);
```

DESCRIPTION

Grid meshes are created by passing RTC_GEOMETRY_TYPE_GRID to the rtcNewGeometry function call, and contain an array of grid primitives. This array of grids can be specified by setting up a grid buffer (with RTC_BUFFER_TYPE_GRID type and RTC_FORMAT_GRID format) and the grid mesh vertices by setting a vertex buffer (RTC_BUFFER_TYPE_VERTEX type). See rtcSetGeometryBuffer and rtcSetSharedGeometryBuffer for more details on how to set buffers. The number of grid primitives in the grid mesh is inferred from the size of the grid buffer.

The vertex buffer contains an array of single precision x, y, z floating point coordinates (RTC_FORMAT_FLOAT3 format), and the number of vertices is inferred from the size of that buffer.

Each grid in the grid buffer is of the type RTCGrid:

```
struct RTCGrid
{
  unsigned int startVertexID;
  unsigned int stride;
  unsigned short width,height;
};
```

The RTCGrid structure describes a 2D grid of vertices (with respect to the vertex buffer of the grid mesh). The width and height members specify the number of vertices in u and v direction, e.g. setting both width and height to 3 sets up a 3×3 vertex grid. The maximum allowed width and height is 32767. The startVertexID specifies the ID of the top-left vertex in the vertex grid, while the stride parameter specifies a stride (in number of vertices) used to step to the next row.

A vertex grid of dimensions width and height is treated as a (width-1) x (height-1) grid of quads (triangle-pairs), with the same shared edge handling as for regular quad meshes. However, the u/v coordinates have the uniform range [0..1] for an entire vertex grid. The u direction follows the width of the grid while the v direction the height.

For multi-segment motion blur, the number of time steps must be first specified using the rtcSetGeometryTimeStepCount call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size.

EXIT STATUS

On failure NULL is returned and an error code is set that can be queried using rtcGetDeviceError.

SEE ALSO

rtcNewGeometry

7.33 RTC_GEOMETRY_TYPE_SUBDIVISION

NAME

RTC_GEOMETRY_TYPE_SUBDIVISION - subdivision geometry type

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_SUBDIVISION);
```

DESCRIPTION

Catmull-Clark subdivision meshes are supported, including support for edge creases, vertex creases, holes, non-manifold geometry, and face-varying interpolation. The number of vertices per face can be in the range of 3 to 15 vertices (triangles, quadrilateral, pentagons, etc).

Subdivision meshes are created by passing RTC_GEOMETRY_TYPE_SUBDIVISION to the rtcNewGeometry function. Various buffers need to be set by the application to set up the subdivision mesh. See rtcSetGeometryBuffer and rtcSetSharedGeometryBuffer for more details on how to set buffers. The face buffer (RTC_BUFFER_TYPE_FACE type and RTC_FORMAT_UINT format) contains the number of edges/indices of each face (3 to 15), and the number of faces is inferred from the size of this buffer. The index buffer (RTC_BUFFER_TYPE_INDEX type) contains multiple (3 to 15) 32-bit vertex indices (RTC_FORMAT_UINT format) for each face, and the number of edges is inferred from the size of this buffer. The vertex buffer (RTC_BUFFER_TYPE_VERTEX type) stores an array of single precision x, y, z floating point coordinates (RTC_FORMAT_FLOAT3 format), and the number of vertices is inferred from the size of this buffer.

Optionally, the application may set additional index buffers using different buffer slots if multiple topologies are required for face-varying interpolation. The standard vertex buffers (RTC_BUFFER_TYPE_VERTEX) are always bound to the geometry topology (topology 0) thus use RTC_BUFFER_TYPE_INDEX with buffer slot 0. User vertex data interpolation may use different topologies as described later.

Optionally, the application can set up the hole buffer (RTC_BUFFER_TYPE_HOLE) which contains an array of 32-bit indices (RTC_FORMAT_UINT format) of faces that should be considered non-existing in all topologies. The number of holes is inferred from the size of this buffer.

Optionally, the application can fill the level buffer (RTC_BUFFER_TYPE_LEVEL) with a tessellation rate for each of the edges of each face. This buffer must have the same size as the index buffer. The tessellation level is a positive floating point value (RTC_FORMAT_FLOAT format) that specifies how many quads along the edge should be 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 are clamped to that value. Note that edges may be shared between (typically 2) faces. To guarantee a watertight tessellation, the level of these shared edges should be identical. A uniform tessellation rate for an entire subdivision mesh can be set by using the rtcSetGeometryTessellationRate function. The existence of a level buffer has precedence over the uniform tessellation rate.

Optionally, the application can fill the sparse edge crease buffers to make edges appear sharper. The edge crease index buffer (RTC_BUFFER_TYPE_EDGE_CREASE_INDEX) contains an array of pairs of 32-bit vertex indices (RTC_FORMAT_UINT2 format) that specify unoriented edges in the geometry topology. The edge crease weight buffer (RTC_BUFFER_TYPE_EDGE_CREASE_WEIGHT) stores for each of these crease edges a positive floating point weight (RTC_FORMAT_FLOAT

format). The number of edge creases is inferred from the size of these buffers, which has to be identical. The larger a 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) do not have to be stored, as both are considered the same unoriented edge. Edge crease features are shared between all topologies.

Optionally, the application can fill the sparse vertex crease buffers to make vertices appear sharper. The vertex crease index buffer (`RTC_BUFFER_TYPE_VERTEX_CREASE_INDEX`), contains an array of 32-bit vertex indices (`RTC_FORMAT_UINT` format) to specify a set of vertices from the geometry topology. The vertex crease weight buffer (`RTC_BUFFER_TYPE_VERTEX_CREASE_WEIGHT`) specifies for each of these vertices a positive floating point weight (`RTC_FORMAT_FLOAT` format). The number of vertex creases is inferred from the size of these buffers, and has to be identical. The larger a 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 shared between all topologies.

Subdivision modes can be used to force linear interpolation for parts of the subdivision mesh; see `rtcSetGeometrySubdivisionMode` for more details.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers have to have the same stride and size.

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

Parametrization

The parametrization for subdivision faces is different for quadrilaterals and non-quadrilateral faces.

The parametrization of a quadrilateral face uses the first vertex `p0` as base point, and the vector `p1 - p0` as u-direction and `p3 - p0` as v-direction.

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

```
unsigned int l = floorf(0.5f*U);
unsigned int h = floorf(0.5f*V);
unsigned int i = 4*h+l;
float u = 2.0f*fracf(0.5f*U)-0.5f;
float v = 2.0f*fracf(0.5f*V)-0.5f;
```

This encoding allows local subpatch UVs to be in the range `[-0.5,1.5[` thus negative subpatch UVs can be passed to `rtcInterpolate` to sample subpatches slightly out of bounds. This can be useful to calculate derivatives using finite differences if required. The encoding further has the property that one can just move the value `u` (or `v`) on a subpatch by adding `du` (or `dv`) to the special UV encoding as long as it does not fall out of the `[-0.5,1.5[` range.

To smoothly interpolate vertex attributes over the subdivision surface we recommend using the `rtcInterpolate` function, which will apply the standard subdivision rules for interpolation and automatically takes care of the special UV encoding for non-quadrilaterals.

Face-Varying Data

Face-varying interpolation is supported through multiple topologies per subdivision mesh and binding such topologies to vertex attribute 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 `i` has a separate index buffer (specified using `RTC_BUFFER_TYPE_INDEX` with buffer slot `i`) and separate subdivision mode that can be set using `rtcSetGeometrySubdivisionMode`. A vertex attribute buffer `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE` bound to a buffer slot `j` can be assigned to use a topology for interpolation using the `rtcSetGeometryVertexAttributeTopology` call.

The face buffer (`RTC_BUFFER_TYPE_FACE` type) 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.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

7.34 RTC_GEOMETRY_TYPE_CURVE

NAME

```
RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE -
  flat curve geometry with linear basis

RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE -
  flat curve geometry with cubic Bézier basis

RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE -
  flat curve geometry with cubic B-spline basis

RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE -
  flat curve geometry with cubic Hermite basis

RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE -
  flat curve geometry with Catmull-Rom basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE -
  flat normal oriented curve geometry with cubic Bézier basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE -
  flat normal oriented curve geometry with cubic B-spline basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE -
  flat normal oriented curve geometry with cubic Hermite basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE -
  flat normal oriented curve geometry with Catmull-Rom basis

RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE -
  capped cone curve geometry with linear basis - discontinuous at edge boundaries

RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE -
  capped cone curve geometry with linear basis and spherical ending

RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE -
  swept surface curve geometry with cubic Bézier basis

RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE -
  swept surface curve geometry with cubic B-spline basis

RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE -
  swept surface curve geometry with cubic Hermite basis

RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE -
  swept surface curve geometry with Catmull-Rom basis
```

SYNOPSIS

```
#include <embree4/rtcore.h>

rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE);
```

```
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE);
```

DESCRIPTION

Curves with per vertex radii are supported with linear, cubic Bézier, cubic B-spline, and cubic Hermite bases. Such curve geometries are created by passing `RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE`, `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT_BEZIER_CURVE`, `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT_BSPLINE_CURVE`, `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT_HERMITE_CURVE`, `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT_CATMULL_ROM_CURVE`, `RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE`, or `RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE` to the `rtcNewGeometry` function. The curve indices can be specified through an index buffer (`RTC_BUFFER_TYPE_INDEX`) and the curve vertices through a vertex buffer (`RTC_BUFFER_TYPE_VERTEX`). For the Hermite basis a tangent buffer (`RTC_BUFFER_TYPE_TANGENT`), normal oriented curves a normal buffer (`RTC_BUFFER_TYPE_NORMAL`), and for normal oriented Hermite curves a normal derivative buffer (`RTC_BUFFER_TYPE_NORMAL_DERIVATIVE`) has to get specified additionally. See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers.

The index buffer contains an array of 32-bit indices (`RTC_FORMAT_UINT` format), each pointing to the first control vertex in the vertex buffer, but also to the first tangent in the tangent buffer, and first normal in the normal buffer if these buffers are present.

The vertex buffer stores each control vertex in the form of a single precision position and radius stored in (`x`, `y`, `z`, `r`) order in memory (`RTC_FORMAT_FLOAT4` format). The number of vertices is inferred from the size of this buffer. The radii may be smaller than zero but the interpolated radii should always be greater or equal to zero. Similarly, the tangent buffer stores the derivative of each control vertex (`x`, `y`, `z`, `r` order and `RTC_FORMAT_FLOAT4` format) and the normal buffer stores a single precision normal per control vertex (`x`, `y`, `z` order and `RTC_FORMAT_FLOAT3` format).

Linear Basis

For the linear basis the indices point to the first of 2 consecutive control points in the vertex buffer. The first control point is the start and the second control point the end of the line segment. When constructing hair strands in this basis, the end-point can be shared with the start of the next line segment.

For the linear basis the user optionally can provide a flags buffer of type `RTC_BUFFER_TYPE_FLAGS` which contains bytes that encode if the left neighbor segment (`RTC_CURVE_FLAG_NEIGHBOR_LEFT` flag) and/or right neighbor segment (`RTC_CURVE_FLAG_NEIGHBOR_RIGHT` flags) exist (see RTCCurveFlags). If this buffer is not set, than the left/right neighbor bits are automatically calculated

base on the index buffer (left segment exists if segment(id-1)+1 == segment(id) and right segment exists if segment(id+1)-1 == segment(id)).

A left neighbor segment is assumed to end at the start vertex of the current segment, and to start at the previous vertex in the vertex buffer. Similarly, the right neighbor segment is assumed to start at the end vertex of the current segment, and to end at the next vertex in the vertex buffer.

Only when the left and right bits are properly specified the current segment can properly attach to the left and/or right neighbor, otherwise the touching area may not get rendered properly.

Bézier Basis For the cubic Bézier basis the indices point to the first of 4 consecutive control points in the vertex buffer. These control points use the cubic Bézier basis, where the first control point represents the start point of the curve, and the 4th control point the end point of the curve. The Bézier basis is interpolating, thus the curve does go exactly through the first and fourth control vertex.

B-spline Basis For the cubic B-spline basis the indices point to the first of 4 consecutive control points in the vertex buffer. These control points make up a cardinal cubic B-spline (implicit equidistant knot vector). This basis is not interpolating, thus the curve does in general not go through any of the control points directly. A big advantage of this basis is that 3 control points can be shared for two continuous neighboring curve segments, e.g. the curves (p0,p1,p2,p3) and (p1,p2,p3,p4) are C1 continuous. This feature makes this basis a good choice to construct continuous multi-segment curves, as memory consumption can be kept minimal.

Hermite Basis For the cubic Hermite basis the indices point to the first of 2 consecutive points in the vertex buffer, and the first of 2 consecutive tangents in the tangent buffer. These two points and two tangents make up a cubic Hermite curve. This basis is interpolating, thus does exactly go through the first and second control point, and the first order derivative at the begin and end matches exactly the value specified in the tangent buffer. When connecting two segments continuously, the end point and tangent of the previous segment can be shared. Different versions of Catmull-Rom splines can be easily constructed using the Hermite basis, by calculating a proper tangent buffer from the control points.

Catmull-Rom Basis For the Catmull-Rom basis the indices point to the first of 4 consecutive control points in the vertex buffer. This basis goes through p1 and p2, with tangents (p2-p0)/2 and (p3-p1)/2.

Flat Curves The `RTC_GEOMETRY_TYPE_FLAT_*` flat mode is a fast mode designed to render distant hair. In this mode the curve is rendered as a connected sequence of ray facing quads. Individual quads are considered to have subpixel size, and zooming onto the curve might show geometric artifacts. The number of quads to subdivide into can be specified through the `rtcSetGeometryTessellationRate` function. By default the tessellation rate is 4.

Normal Oriented Curves The `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_*` mode is a mode designed to render blades of grass. In this mode a vertex spline has to get specified as for the previous modes, but additionally a normal spline is required. If the Hermite basis is used, the `RTC_BUFFER_TYPE_NORMAL` and `RTC_BUFFER_TYPE_NORMAL_DERIVATIVE` buffers have both to be set.

The curve is rendered as a flat band whose center approximately follows the provided vertex spline, whose half width approximately follows the provided

radius spline, and whose normal orientation approximately follows the provided normal spline.

To intersect the normal oriented curve, we perform a newton-raphson style intersection of a ray with a tensor product surface of a linear basis (perpendicular to the curve) and cubic Bézier basis (along the curve). We use a guide curve and its derivatives to construct the control points of that surface. The guide curve is defined by a sweep surface defined by sweeping a line centered at the vertex spline location along the curve. At each parameter value the half width of the line matches the radius spline, and the direction matches the cross product of the normal from the normal spline and tangent of the vertex spline. Note that this construction does not work when the provided normals are parallel to the curve direction. For this reason the provided normals should best be kept as perpendicular to the curve direction as possible. We further assume second order derivatives of the center curve to be zero for this construction, as otherwise very large curvatures occurring in corner cases, can thicken the constructed curve significantly.

Round Curves In the `RTC_GEOMETRY_TYPE_ROUND_*` round mode, a real geometric surface is rendered for the curve, which is more expensive but allows closeup views.

For the linear basis the round mode renders a cone that tangentially touches a start-sphere and end-sphere. The start sphere is rendered when no previous segments is indicated by the neighbor bits. The end sphere is always rendered but parts that lie inside the next segment are clipped away (if that next segment exists). This way a curve is closed on both ends and the interior will render properly as long as only neighboring segments penetrate into a segment. For this to work properly it is important that the flags buffer is properly populated with neighbor information.

For the cubic polynomial bases, the round mode renders a sweep surface by sweeping a varying radius circle tangential along the curve. As a limitation, the radius of the curve has to be smaller than the curvature radius of the curve at each location on the curve.

The intersection with the curve segment stores the parametric hit location along the curve segment as u-coordinate (range 0 to +1).

For flat curves, the v-coordinate is set to the normalized distance in the range -1 to +1. For normal oriented curves the v-coordinate is in the range 0 to 1. For the linear basis and in round mode the v-coordinate is set to zero.

In flat mode, the geometry normal `Ng` is set to the tangent of the curve at the hit location. In round mode and for normal oriented curves, the geometry normal `Ng` is set to the non-normalized geometric normal of the surface.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size. For the Hermite basis also a tangent buffer has to be set for each time step and for normal oriented curves a normal buffer has to get specified for each time step.

Also see tutorials Hair and Curves for examples of how to create and use curve geometries.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, RTCCurveFlags

7.35 RTC_GEOMETRY_TYPE_POINT

NAME

```
RTC_GEOMETRY_TYPE_SPHERE_POINT -
  point geometry spheres

RTC_GEOMETRY_TYPE_DISC_POINT -
  point geometry with ray-oriented discs

RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT -
  point geometry with normal-oriented discs
```

SYNOPSIS

```
#include <embree4/rtcore.h>

rtcNewGeometry(device, RTC_GEOMETRY_TYPE_SPHERE_POINT);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_DISC_POINT);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT);
```

DESCRIPTION

Points with per vertex radii are supported with sphere, ray-oriented discs, and normal-oriented discs geometric representations. Such point geometries are created by passing `RTC_GEOMETRY_TYPE_SPHERE_POINT`, `RTC_GEOMETRY_TYPE_DISC_POINT`, or `RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT` to the `rtcNewGeometry` function. The point vertices can be specified t through a vertex buffer (`RTC_BUFFER_TYPE_VERTEX`). For the normal oriented discs a normal buffer (`RTC_BUFFER_TYPE_NORMAL`) has to get specified additionally. See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers.

The vertex buffer stores each control vertex in the form of a single precision position and radius stored in (`x`, `y`, `z`, `r`) order in memory (`RTC_FORMAT_FLOAT4` format). The number of vertices is inferred from the size of this buffer. Similarly, the normal buffer stores a single precision normal per control vertex (`x`, `y`, `z` order and `RTC_FORMAT_FLOAT3` format).

In the `RTC_GEOMETRY_TYPE_SPHERE_POINT` mode, a real geometric surface is rendered for the curve, which is more expensive but allows closeup views.

The `RTC_GEOMETRY_TYPE_DISC_POINT` flat mode is a fast mode designed to render distant points. In this mode the point is rendered as a ray facing disc.

The `RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT` mode is a mode designed as a midpoint geometrically between ray facing discs and spheres. In this mode the point is rendered as a normal oriented disc.

For all point types, only the hit distance and geometry normal is returned as hit information, u and v are set to zero.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size.

Also see tutorial [Points] for an example of how to create and use point geometries.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

7.36 RTC_GEOMETRY_TYPE_USER

NAME

`RTC_GEOMETRY_TYPE_USER - user geometry type`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_USER);
```

DESCRIPTION

User-defined geometries contain a number of user-defined primitives, just like triangle meshes contain multiple triangles. The shape of the user-defined primitives is specified through registered callback functions, which enable extending Embree with arbitrary types of primitives.

User-defined geometries are created by passing `RTC_GEOMETRY_TYPE_USER` to the `rtcNewGeometry` function call. One has to set the number of primitives (see `rtcSetGeometryUserPrimitiveCount`), a user data pointer (see `rtcSetGeometryUserData`), a bounding function closure (see `rtcSetGeometryBoundsFunction`), as well as user-defined intersect (see `rtcSetGeometryIntersectFunction`) and occluded (see `rtcSetGeometryOccludedFunction`) callback functions. The bounding function is used to query the bounds of all time steps of a user primitive, while the intersect and occluded callback functions are called to intersect the primitive with a ray. The user data pointer is passed to each callback invocation and can be used to point to the application's representation of the user geometry.

The creation of a user geometry typically looks the following:

```
RTCGeometry geometry = rtcNewGeometry(device, RTC_GEOMETRY_TYPE_USER);
rtcSetGeometryUserPrimitiveCount(geometry, numPrimitives);
rtcSetGeometryUserData(geometry, userGeometryRepresentation);
rtcSetGeometryBoundsFunction(geometry, boundsFunction);
rtcSetGeometryIntersectFunction(geometry, intersectFunction);
rtcSetGeometryOccludedFunction(geometry, occludedFunction);
```

Please have a look at the `rtcSetGeometryBoundsFunction`, `rtcSetGeometryIntersectFunction`, and `rtcSetGeometryOccludedFunction` functions on the implementation of the callback functions.

Primitives of a user geometry are ignored during rendering when their bounds are empty, thus bounds have lower>upper in at least one dimension.

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

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcSetGeometryUserPrimitiveCount, rtcSetGeometryUserData, rtcSetGeometryBoundsFunction, rtcSetGeometryIntersectFunction, rtcSetGeometryOccludedFunction

7.37 RTC_GEOMETRY_TYPE_INSTANCE

NAME

`RTC_GEOMETRY_TYPE_INSTANCE - instance geometry type`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCGeometry geometry =
   rtcNewGeometry(device, RTC_GEOMETRY_TYPE_INSTANCE);
```

DESCRIPTION

Embree supports instancing of scenes using affine transformations (3×3 matrix plus translation). As the instanced scene is stored only a single time, even if instanced to multiple locations, this feature can be used to create very complex scenes with small memory footprint.

Embree supports both single-level instancing and multi-level instancing. The maximum instance nesting depth is `RTC_MAX_INSTANCE_LEVEL_COUNT`; it can be configured at compile-time using the constant `EMBREE_MAX_INSTANCE_LEVEL_COUNT`. Users should adapt this constant to their needs: instances nested any deeper are silently ignored in release mode, and cause assertions in debug mode.

Instances are created by passing `RTC_GEOMETRY_TYPE_INSTANCE` to the `rtcNewGeometry` function call. The instanced scene can be set using the `rtcSetGeometryInstancedScene` call, and the affine transformation can be set using the `rtcSetGeometryTransform` function.

Please note that `rtcCommitScene` on the instanced scene should be called first, followed by `rtcCommitGeometry` on the instance, followed by `rtcCommitScene` for the top-level scene containing the instance.

If a ray hits the instance, the `geomID` and `primID` members of the hit are set to the geometry ID and primitive ID of the hit primitive in the instanced scene, and the `instID` member of the hit is set to the geometry ID of the instance in the top-level scene.

The instancing scheme can also be implemented using user geometries. To achieve this, the user geometry code should set the `instID` member of the ray query context to the geometry ID of the instance, then trace the transformed ray, and finally set the `instID` field of the ray query context again to -1. The `instID` field is copied automatically by each primitive intersector into the `instID` field of the hit structure when the primitive is hit. See the User Geometry tutorial for an example.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` function. Then a transformation for each time step can be specified using the `rtcSetGeometryTransform` function.

See tutorials Instanced Geometry and Multi Level Instancing for examples of how to use instances.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcSetGeometryInstancedScene, rtcSetGeometryTransform

7.38 RTCCurveFlags

NAME

RTCCurveFlags - per segment flags for curve geometry

SYNOPSIS

```
#include <embree4/rtcore.h>

enum RTCCurveFlags
{
  RTC_CURVE_FLAG_NEIGHBOR_LEFT  = (1 << 0),
  RTC_CURVE_FLAG_NEIGHBOR_RIGHT = (1 << 1)
};
```

DESCRIPTION

The RTCCurveFlags type is used for linear curves to determine if the left and/or right neighbor segment exist. Therefore one attaches a buffer of type RTC_BUFFER_TYPE_FLAGS to the curve geometry which stores an individual byte per curve segment.

If the RTC_CURVE_FLAG_NEIGHBOR_LEFT flag in that byte is enabled for a curve segment, then the left segment exists (which starts one vertex before the start vertex of the current curve) and the current segment is rendered to properly attach to that segment.

If the RTC_CURVE_FLAG_NEIGHBOR_RIGHT flag in that byte is enabled for a curve segment, then the right segment exists (which ends one vertex after the end vertex of the current curve) and the current segment is rendered to properly attach to that segment.

When not properly specifying left and right flags for linear curves, the rendering at the ending of these curves may not look correct, in particular when round linear curves are viewed from the inside.

EXIT STATUS

SEE ALSO

RTC_GEOMETRY_TYPE_CURVE

7.39 rtcRetainGeometry

NAME

`rtcRetainGeometry - increments the geometry reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcRetainGeometry(RTCGeometry geometry);
```

DESCRIPTION

Geometry objects are reference counted. The `rtcRetainGeometry` function increments the reference count of the passed geometry object (`geometry` argument). This function together with `rtcReleaseGeometry` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcReleaseGeometry

7.40 rtcReleaseGeometry

NAME

`rtcReleaseGeometry - decrements the geometry reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcReleaseGeometry(RTCGeometry geometry);
```

DESCRIPTION

Geometry objects are reference counted. The `rtcReleaseGeometry` function decrements the reference count of the passed geometry object (`geometry` argument). When the reference count falls to 0, the geometry gets destroyed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcRetainGeometry

7.41 rtcCommitGeometry

NAME

`rtcCommitGeometry - commits geometry changes`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcCommitGeometry(RTCGeometry geometry);
```

DESCRIPTION

The `rtcCommitGeometry` function is used to commit all geometry changes performed to a geometry (`geometry` parameter). After a geometry gets modified, this function must be called to properly update the internal state of the geometry to perform interpolations using `rtcInterpolate` or to commit a scene containing the geometry using `rtcCommitScene`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcInterpolate, rtcCommitScene

7.42 rtcEnableGeometry

NAME

`rtcEnableGeometry - enables the geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcEnableGeometry(RTCGeometry geometry);
```

DESCRIPTION

The `rtcEnableGeometry` function enables the specified geometry (`geometry` argument). Only enabled geometries are rendered. Each geometry is enabled by default at construction time.

After enabling a geometry, the scene containing that geometry must be committed using `rtcCommitScene` for the change to have effect.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcDisableGeometry, rtcCommitScene

7.43 rtcDisableGeometry

NAME

`rtcDisableGeometry - disables the geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcDisableGeometry(RTCGeometry geometry);
```

DESCRIPTION

The `rtcDisableGeometry` function disables the specified geometry (`geometry` argument). A disabled geometry is not rendered. Each geometry is enabled by default at construction time.

After disabling a geometry, the scene containing that geometry must be committed using `rtcCommitScene` for the change to have effect.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcEnableGeometry, rtcCommitScene

7.44 rtcSetGeometryTimeStepCount

NAME

`rtcSetGeometryTimeStepCount - sets the number of time steps of the geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryTimeStepCount(
  RTCGeometry geometry,
  unsigned int timeStepCount
);
```

DESCRIPTION

The `rtcSetGeometryTimeStepCount` function sets the number of time steps for multi-segment motion blur (`timeStepCount` parameter) of the specified geometry (`geometry` parameter).

For triangle meshes (`RTC_GEOMETRY_TYPE_TRIANGLE`), quad meshes (`RTC_GEOMETRY_TYPE_QUAD`), curves (`RTC_GEOMETRY_TYPE_CURVE`), points (`RTC_GEOMETRY_TYPE_POINT`), and subdivision geometries (`RTC_GEOMETRY_TYPE_SUBDIVISION`), the number of time steps directly corresponds to the number of vertex buffer slots available (`RTC_BUFFER_TYPE_VERTEX` buffer type). For these geometries, one vertex buffer per time step must be specified when creating multi-segment motion blur geometries.

For instance geometries (`RTC_GEOMETRY_TYPE_INSTANCE`), a transformation must be specified for each time step (see `rtcSetGeometryTransform`).

For user geometries, the registered bounding callback function must provide a bounding box per primitive and time step, and the intersection and occlusion callback functions should properly intersect the motion-blurred geometry at the ray time.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcSetGeometryTimeRange

7.45 rtcSetGeometryTimeRange

NAME

`rtcSetGeometryTimeRange - sets the time range for a motion blur geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryTimeRange(
  RTCGeometry geometry,
  float startTime,
  float endTime
);
```

DESCRIPTION

The `rtcSetGeometryTimeRange` function sets a time range which defines the start (and end time) of the first (and last) time step of a motion blur geometry. The time range is defined relative to the camera shutter interval [0,1] but it can be arbitrary. Thus the startTime can be smaller, equal, or larger 0, indicating a geometry whose animation definition start before, at, or after the camera shutter opens. Similar the endTime can be smaller, equal, or larger than 1, indicating a geometry whose animation definition ends after, at, or before the camera shutter closes. The startTime has to be smaller or equal to the endTime.

The default time range when this function is not called is the entire camera shutter [0,1]. For best performance at most one time segment of the piece wise linear definition of the motion should fall outside the shutter window to the left and to the right. Thus do not set the startTime or endTime too far outside the [0,1] interval for best performance.

This time range feature will also allow geometries to appear and disappear during the camera shutter time if the specified time range is a sub range of [0,1].

Please also have a look at the `rtcSetGeometryTimeStepCount` function to see how to define the time steps for the specified time range.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryTimeStepCount

7.46 rtcSetGeometryVertexAttributeCount

NAME

```
rtcSetGeometryVertexAttributeCount - sets the number of vertex
  attributes of the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryVertexAttributeCount(
  RTCGeometry geometry,
  unsigned int vertexAttributeCount
);
```

DESCRIPTION

The `rtcSetGeometryVertexAttributeCount` function sets the number of slots (`vertexAttributeCount` parameter) for vertex attribute buffers (`RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`) that can be used for the specified geometry (`geometry` parameter).

This function is supported only for triangle meshes (`RTC_GEOMETRY_TYPE_TRIANGLE`), quad meshes (`RTC_GEOMETRY_TYPE_QUAD`), curves (`RTC_GEOMETRY_TYPE_CURVE`), points (`RTC_GEOMETRY_TYPE_POINT`), and subdivision geometries (`RTC_GEOMETRY_TYPE_SUBDIVISION`).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, RTCBufferType

7.47 rtcSetGeometryMask

NAME

`rtcSetGeometryMask - sets the geometry mask`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryMask(
  RTCGeometry geometry,
  unsigned int mask
);
```

DESCRIPTION

The `rtcSetGeometryMask` function sets a 32-bit geometry mask (`mask` argument) for the specified geometry (`geometry` argument).

This geometry mask is used together with the ray mask stored inside the `mask` field of the ray. The primitives of the geometry are hit by the ray only if the bitwise and operation of the geometry mask with the ray mask is not 0. This feature can be used to disable selected geometries for specifically tagged rays, e.g. to disable shadow casting for certain geometries.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTCRay, rtcGetDeviceProperty

7.48 rtcSetGeometryBuildQuality

NAME

`rtcSetGeometryBuildQuality - sets the build quality for the geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryBuildQuality(
  RTCGeometry geometry,
  enum RTCBuildQuality quality
);
```

DESCRIPTION

The `rtcSetGeometryBuildQuality` function sets the build quality (`quality` argument) for the specified geometry (`geometry` argument). The per-geometry build quality is only a hint and may be ignored. Embree currently uses the per-geometry build quality when the scene build quality is set to `RTC_BUILD_QUALITY_LOW`. In this mode a two-level acceleration structure is build, and geometries build a separate acceleration structure using the geometry build quality. The per-geometry build quality can be one of:

- `RTC_BUILD_QUALITY_LOW`: Creates lower quality data structures, e.g. for dynamic scenes.
- `RTC_BUILD_QUALITY_MEDIUM`: Default build quality for most usages. Gives a good compromise between build and render performance.
- `RTC_BUILD_QUALITY_HIGH`: Creates higher quality data structures for final-frame rendering. Enables a spatial split builder for certain primitive types.
- `RTC_BUILD_QUALITY_REFIT`: Uses a BVH refitting approach when changing only the vertex buffer.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetSceneBuildQuality

7.49 rtcSetGeometryMaxRadiusScale

NAME

`rtcSetGeometryMaxRadiusScale - assigns a maximal curve radius scale factor for min-width feature`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryMaxRadiusScale(RTCGeometry geometry, float maxRadiusScale);
```

DESCRIPTION

The `rtcSetMaxGeometryScale` function specifies a maximal scaling factor for curve radii used by the min-width feature.

The min-width feature can increase the radius of curves and points, in order to reduce aliasing and improve render times. The feature is disabled by default and has to get enabled using the EMBREE_MIN_WIDTH cmake option.

To use the feature, one has to specify a maximal curve radius scaling factor using the rtcSetGeometryMaxRadiusScale function. This factor should be a small number (e.g. 4) as the constructed BVH bounds get increased in order to bound the curve in the worst case of maximal radii.

One also has to set the minWidthDistanceFactor in the RTCRayQueryContext when tracing a ray. This factor controls the target radius size of a curve or point at some distance away of the ray origin.

For each control point p with radius r of a curve or point primitive, the primitive intersectors first calculate a target radius r' as:

```
r' = length(p-ray_org) * minWidthDistanceFactor
```

Typically the minWidthDistanceFactor is set by the application such that the target radius projects to the width of half a pixel (thus primitive diameter is pixel sized).

The target radius r' is then clamped against the minimal bound r and maximal bound maxRadiusScale*r to obtain the final radius r'':

```
r'' = max(r, min(r', maxRadiusScale*r))
```

Thus curves or points close to the camera are rendered with a normal radii r, and curves or points far from the camera are not enlarged too much, as this would be very expensive to render.

When `rtcSetGeometryMaxRadiusScale` function is not invoked for a curve or point geometry (or if the maximal scaling factor is set to 1.0), then the curve or point geometry renders normally, with radii not modified by the min-width feature.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcInitRayQueryContext

7.50 rtcSetGeometryBuffer

NAME

`rtcSetGeometryBuffer - assigns a view of a buffer to the geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot,
  enum RTCFormat format,
  RTCBuffer buffer,
  size_t byteOffset,
  size_t byteStride,
  size_t itemCount
);
```

DESCRIPTION

The `rtcSetGeometryBuffer` function binds a view of a buffer object (`buffer` argument) to a geometry buffer type and slot (`type` and `slot` argument) of the specified geometry (`geometry` argument).

One can specify the start of the first buffer element in bytes (`byteOffset` argument), the byte stride between individual buffer elements (`byteStride` argument), the format of the buffer elements (`format` argument), and the number of elements to bind (`itemCount`).

The start address (`byteOffset` argument) and stride (`byteStride` argument) must be both aligned to 4 bytes, otherwise the `rtcSetGeometryBuffer` function will fail.

After successful completion of this function, the geometry will hold a reference to the buffer object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetSharedGeometryBuffer, rtcSetNewGeometryBuffer

7.51 rtcSetSharedGeometryBuffer

NAME

`rtcSetSharedGeometryBuffer - assigns a view of a shared data buffer to a geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetSharedGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot,
  enum RTCFormat format,
  const void* ptr,
  size_t byteOffset,
  size_t byteStride,
  size_t itemCount
);
```

DESCRIPTION

The `rtcSetSharedGeometryBuffer` function binds a view of a shared user-managed data buffer (`ptr` argument) to a geometry buffer type and slot (`type` and `slot` argument) of the specified geometry (`geometry` argument).

One can specify the start of the first buffer element in bytes (`byteOffset` argument), the byte stride between individual buffer elements (`byteStride` argument), the format of the buffer elements (`format` argument), and the number of elements to bind (`itemCount`).

The start address (`byteOffset` argument) and stride (`byteStride` argument) must be both aligned to 4 bytes; otherwise the `rtcSetSharedGeometryBuffer` function will fail.

When the buffer will be used as a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` and `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`), the last buffer element must be readable using 16-byte SSE load instructions, thus padding the last element is required for certain layouts. E.g. a standard `float3` vertex buffer layout should add storage for at least one more float to the end of the buffer.

The buffer data must remain valid for as long as the buffer may be used, and the user is responsible for freeing the buffer data when no longer required.

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

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuffer, rtcSetNewGeometryBuffer

7.52 rtcSetNewGeometryBuffer

NAME

```
rtcSetNewGeometryBuffer - creates and assigns a new data buffer to
  the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void* rtcSetNewGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot,
  enum RTCFormat format,
  size_t byteStride,
  size_t itemCount
);
```

DESCRIPTION

The `rtcSetNewGeometryBuffer` function creates a new data buffer of specified format (`format` argument), byte stride (`byteStride` argument), and number of items (`itemCount` argument), and assigns it to a geometry buffer slot (`type` and `slot` argument) of the specified geometry (`geometry` argument). The buffer data is managed internally and automatically freed when the geometry is destroyed.

The byte stride (`byteStride` argument) must be aligned to 4 bytes; otherwise the `rtcSetNewGeometryBuffer` function will fail.

The allocated buffer will be automatically over-allocated slightly when used as a vertex buffer, where a requirement is that each buffer element should be readable using 16-byte SSE load instructions.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuffer, rtcSetSharedGeometryBuffer

7.53 RTCFormat

NAME

RTCFormat - specifies format of data in buffers

SYNOPSIS

```
#include <embree4/rtcore_ray.h>

enum RTCFormat
{
  RTC_FORMAT_UINT,
  RTC_FORMAT_UINT2,
  RTC_FORMAT_UINT3,
  RTC_FORMAT_UINT4,

  RTC_FORMAT_FLOAT,
  RTC_FORMAT_FLOAT2,
  RTC_FORMAT_FLOAT3,
  RTC_FORMAT_FLOAT4,
  RTC_FORMAT_FLOAT5,
  RTC_FORMAT_FLOAT6,
  RTC_FORMAT_FLOAT7,
  RTC_FORMAT_FLOAT8,
  RTC_FORMAT_FLOAT9,
  RTC_FORMAT_FLOAT10,
  RTC_FORMAT_FLOAT11,
  RTC_FORMAT_FLOAT12,
  RTC_FORMAT_FLOAT13,
  RTC_FORMAT_FLOAT14,
  RTC_FORMAT_FLOAT15,
  RTC_FORMAT_FLOAT16,

  RTC_FORMAT_FLOAT3X4_ROW_MAJOR,
  RTC_FORMAT_FLOAT4X4_ROW_MAJOR,

  RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR,
  RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR,

  RTC_FORMAT_GRID,
};
```

DESCRIPTION

The `RTFormat` structure defines the data format stored in data buffers provided to Embree using the rtcSetGeometryBuffer, rtcSetSharedGeometryBuffer, and rtcSetNewGeometryBuffer API calls.

The `RTC_FORMAT_UINT/2/3/4` format are used to specify that data buffers store unsigned integers, or unsigned integer vectors of size 2,3 or 4. This format has typically to get used when specifying index buffers, e.g. `RTC_FORMAT_UINT3` for triangle meshes.

The `RTC_FORMAT_FLOAT/2/3/4...` format are used to specify that data buffers store single precision floating point values, or vectors there of (size 2,3,4, etc.). This format is typcally used to specify to format of vertex buffers, e.g. the `RTC_FORMAT_FLOAT3` type for vertex buffers of triangle meshes.

The RTC_FORMAT_FLOAT3X4_ROW_MAJOR and RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR formats, specify a 3x4 floating point matrix layed out either row major or column major. The RTC_FORMAT_FLOAT4X4_ROW_MAJOR and RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR formats, specify a 4x4 floating point matrix layed out either row major or column major. These matrix formats are used in the rtcSetGeometryTransform function in order to set a transformation matrix for geometries.

The RTC_FORMAT_GRID is a special data format used to specify grid primitives of layout RTCGrid when creating grid geometries (see RTC_GEOMETRY_TYPE_GRID).

EXIT STATUS

SEE ALSO

rtcSetGeometryBuffer, rtcSetSharedGeometryBuffer, rtcSetNewGeometryBuffer, rtcSetGeometryTransform

7.54 RTCBufferType

NAME

RTCFormat - specifies format of data in buffers

SYNOPSIS

```
#include <embree4/rtcore_ray.h>

enum RTCBufferType
{
  RTC_BUFFER_TYPE_INDEX            = 0,
  RTC_BUFFER_TYPE_VERTEX           = 1,
  RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE = 2,
  RTC_BUFFER_TYPE_NORMAL           = 3,
  RTC_BUFFER_TYPE_TANGENT          = 4,
  RTC_BUFFER_TYPE_NORMAL_DERIVATIVE = 5,

  RTC_BUFFER_TYPE_GRID                = 8,

  RTC_BUFFER_TYPE_FACE                = 16,
  RTC_BUFFER_TYPE_LEVEL               = 17,
  RTC_BUFFER_TYPE_EDGE_CREASE_INDEX   = 18,
  RTC_BUFFER_TYPE_EDGE_CREASE_WEIGHT  = 19,
  RTC_BUFFER_TYPE_VERTEX_CREASE_INDEX  = 20,
  RTC_BUFFER_TYPE_VERTEX_CREASE_WEIGHT = 21,
  RTC_BUFFER_TYPE_HOLE                = 22,

  RTC_BUFFER_TYPE_FLAGS = 32
};
```

DESCRIPTION

The RTBufferType structure defines slots to assign data buffers to using the rtcSetGeometryBuffer, rtcSetSharedGeometryBuffer, and rtcSetNewGeometryBuffer API calls.

For most geometry types the RTC_BUFFER_TYPE_INDEX slot is used to assign an index buffer, while the RTC_BUFFER_TYPE_VERTEX is used to assign the corresponding vertex buffer.

The RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE slot can get used to assign arbitrary additional vertex data which can get interpolated using the rtcInterpolate API call.

The RTC_BUFFER_TYPE_NORMAL, RTC_BUFFER_TYPE_TANGENT, and RTC_BUFFER_TYPE_NORMAL_DERIVATIVE are special buffers required to assign per vertex normals, tangents, and normal derivatives for some curve types.

The RTC_BUFFER_TYPE_GRID buffer is used to assign the grid primitive buffer for grid geometries (see RTC_GEOMETRY_TYPE_GRID).

The RTC_BUFFER_TYPE_FACE, RTC_BUFFER_TYPE_LEVEL, RTC_BUFFER_TYPE_EDGE_CREASE_INDEX, RTC_BUFFER_TYPE_EDGE_CREASE_WEIGHT, RTC_BUFFER_TYPE_VERTEX_CREASE_INDEX, RTC_BUFFER_TYPE_VERTEX_CREASE_WEIGHT, and RTC_BUFFER_TYPE_HOLE are special buffers required to create subdivision meshes (see RTC_GEOMETRY_TYPE_SUBDIVISION).

The RTC_BUFFER_TYPE_FLAGS can get used to add additional flag per primitive of a geometry, and is currently only used for linear curves.

EXIT STATUS

SEE ALSO

rtcSetGeometryBuffer, rtcSetSharedGeometryBuffer, rtcSetNewGeometryBuffer

7.55 rtcGetGeometryBufferData

NAME

```
rtcGetGeometryBufferData - gets pointer to
  the first buffer view element
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void* rtcGetGeometryBufferData(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot
);
```

DESCRIPTION

The `rtcGetGeometryBufferData` function returns a pointer to the first element of the buffer view attached to the specified buffer type and slot (`type` and `slot` argument) of the geometry (`geometry` argument).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuffer, rtcSetSharedGeometryBuffer, rtcSetNewGeometryBuffer

7.56 rtcUpdateGeometryBuffer

NAME

```
rtcUpdateGeometryBuffer - marks a buffer view bound to the geometry
  as modified
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcUpdateGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot
);
```

DESCRIPTION

The `rtcUpdateGeometryBuffer` function marks the buffer view bound to the specified buffer type and slot (`type` and `slot` argument) of a geometry (`geometry` argument) as modified.

If a data buffer is changed by the application, the `rtcUpdateGeometryBuffer` call must be invoked for that buffer. Each buffer view assigned to a buffer slot is initially marked as modified, thus this function needs to be called only when doing buffer modifications after the first `rtcCommitScene`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, rtcCommitScene

7.57 rtcSetGeometryIntersectFilterFunction

NAME

```
rtcSetGeometryIntersectFilterFunction - sets the intersection filter
  for the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCFilterFunctionNArguments
{
  int* valid;
  void* geometryUserPtr;
  const struct RTCRayQueryContext* context;
  struct RTCRayN* ray;
  struct RTCHitN* hit;
  unsigned int N;
};

typedef void (*RTCFilterFunctionN)(
  const struct RTCFilterFunctionNArguments* args
);

void rtcSetGeometryIntersectFilterFunction(
  RTCGeometry geometry,
  RTCFilterFunctionN filter
);
```

DESCRIPTION

The `rtcSetGeometryIntersectFilterFunction` function registers an intersection filter callback function (`filter` argument) for the specified geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered intersection filter function is invoked for every hit encountered during the `rtcIntersect`-type ray queries and can accept or reject that hit. The feature can be used to define a silhouette for a primitive and reject hits that are outside the silhouette. E.g. a tree leaf could be modeled with an alpha texture that decides whether hit points lie inside or outside the leaf.

If the `RTC_BUILD_QUALITY_HIGH` mode is set, the filter functions may be called multiple times for the same primitive hit. Further, rays hitting exactly the edge might also report two hits for the same surface. For certain use cases, the application may have to work around this limitation by collecting already reported hits (`geomID`/`primID` pairs) and ignoring duplicates.

The filter function callback of type `RTCFilterFunctionN` gets passed a number of arguments through the `RTCFilterFunctionNArguments` structure. The `valid` parameter of that structure points to an integer valid mask (0 means invalid and -1 means valid). The `geometryUserPtr` member is a user pointer optionally set per geometry through the `rtcSetGeometryUserData` function. The `context` member points to the ray query context passed to the ray query function. The `ray` parameter points to `N` rays in SOA layout. The `hit` parameter points to `N` hits in SOA layout to test. The `N` parameter is the number of rays and hits in `ray` and `hit`. The hit distance is provided as the `tfar` value of the ray. If

the hit geometry is instanced, the `instID` member of the ray is valid, and the ray and the potential hit are in object space.

The filter 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. To accept the hit, it just has to leave the valid mask set to `-1`. The filter function is further allowed to change the hit and decrease the `tfar` value of the ray but it should not modify other ray data nor any inactive components of the ray or hit.

When performing ray queries using `rtcIntersect1`, it is guaranteed that the packet size is 1 when the callback is invoked. When performing ray queries using the `rtcIntersect4/8/16` functions, it is not generally guaranteed that the ray packet size (and order of rays inside the packet) passed to the callback matches the initial ray packet. However, under some circumstances these properties are guaranteed, and whether this is the case can be queried using `rtcGetDeviceProperty`.

For many usage scenarios, repacking and re-ordering of rays does not cause difficulties in implementing the callback function. However, algorithms that need to extend the ray with additional data must use the `rayID` component of the ray to identify the original ray to access the per-ray data.

The implementation of the filter function can choose to implement a single code path that uses the ray access helper functions `RTCRay_XXX` and hit access helper functions `RTCHit_XXX` to access ray and hit data. Alternatively the code can branch to optimized implementations for specific sizes of `N` and cast the `ray` and `hit` inputs to the proper packet types.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryOccludedFilterFunction

7.58 rtcSetGeometryOccludedFilterFunction

NAME

```
rtcSetGeometryOccludedFilterFunction - sets the occlusion filter
  for the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryOccludedFilterFunction(
  RTCGeometry geometry,
  RTCFilterFunctionN filter
);
```

DESCRIPTION

The `rtcSetGeometryOccludedFilterFunction` function registers an occlusion filter callback function (`filter` argument) for the specified geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered intersection filter function is invoked for every hit encountered during the `rtcOccluded`-type ray queries and can accept or reject that hit. The feature can be used to define a silhouette for a primitive and reject hits that are outside the silhouette. E.g. a tree leaf could be modeled with an alpha texture that decides whether hit points lie inside or outside the leaf.

Please see the description of the `rtcSetGeometryIntersectFilterFunction` for a description of the filter callback function.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryIntersectFilterFunction

7.59 rtcSetGeometryEnableFilterFunctionFromArguments

NAME

```
rtcSetGeometryEnableFilterFunctionFromArguments - enables
  argument filter functions for the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryEnableFilterFunctionFromArguments(
   RTCGeometry geometry, bool enable);
```

DESCRIPTION

This function enables invokation the filter function passed through `RTCIntersectArguments` or `RTCOccludedArguments` to the intersect and occluded queries. If enable is true the argument filter function invokation is enabled for the geometry or disabled otherwise. By default the invokation of the argument filter function is disabled for some geometry.

The argument filter function invokation can also get enforced for each geometry by using the `RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER` ray query flag that can get passed to `rtcIntersect` and `rtcOccluded` functions. See Section rtcInitIntersectArguments and rtcInitOccludedArguments for more details.

In order to use the argument filter function for some scene, that feature additionally has to get enabled using the `RTC_SCENE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS` scene flag. See Section rtcSetSceneFlags for more details.

EXIT STATUS

On failure an error code is set that can get queried using `rtcGetDeviceError`.

SEE ALSO

rtcInitIntersectArguments, rtcInitOccludedArguments, rtcSetSceneFlags

7.60 rtcInvokeIntersectFilterFromGeometry

NAME

```
rtcInvokeIntersectFilterFromGeometry - invokes the
  intersection filter function from the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcInvokeIntersectFilterFromGeometry(
  const struct RTCIntersectFunctionNArguments* args,
  const struct RTCFilterFunctionNArguments* filterArgs
);
```

DESCRIPTION

The `rtcInvokeIntersectFilterFromGeometry` function can be called inside an `RTCIntersectFunctionN` user geometry callback function to invoke the intersection filter registered to the geometry. For this an `RTCFilterFunctionNArguments` structure must be created (see `rtcSetGeometryIntersectFilterFunction`) which basically consists of a valid mask, a hit packet to filter, the corresponding ray packet, and the packet size. After the invocation of `rtcInvokeIntersectFilterFromGeometry`, only rays that are still valid (valid mask set to -1) should update a hit.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcInvokeOccludedFilterFromGeometry, rtcSetGeometryIntersectFunction

7.61 rtcInvokeOccludedFilterFromGeometry

NAME

```
rtcInvokeOccludedFilterFromGeometry - invokes the occlusion
  filter function from the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcInvokeOccludedFilterFromGeometry(
  const struct RTCOccludedFunctionNArguments* args,
  const struct RTCFilterFunctionNArguments* filterArgs
);
```

DESCRIPTION

The `rtcInvokeOccludedFilterFromGeometry` function can be called inside an `RTCOccludedFunctionN` user geometry callback function to invoke the occlusion filter registered to the geometry. For this an `RTCFilterFunctionNArguments` structure must be created (see `rtcSetGeometryIntersectFilterFunction`) which basically consists of a valid mask, a hit packet to filter, the corresponding ray packet, and the packet size. After the invocation of `rtcInvokeOccludedFilterFromGeometry` only rays that are still valid (valid mask set to -1) should signal an occlusion.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcInvokeIntersectFilterFromGeometry, rtcSetGeometryOccludedFunction

7.62 rtcSetGeometryUserData

NAME

`rtcSetGeometryUserData - sets the user-defined data pointer of the geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryUserData(RTCGeometry geometry, void* userPtr);
```

DESCRIPTION

The `rtcSetGeometryUserData` function sets the user-defined data pointer (`userPtr` argument) for a geometry (`geometry` argument). This user data pointer is intended to be pointing to the application's representation of the geometry, and is passed to various callback functions. The application can use this pointer inside the callback functions to access its geometry representation.

The `rtcGetGeometryUserData` function can be used to query an already set user data pointer of a geometry.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryUserData

7.63 rtcGetGeometryUserData

NAME

```
rtcGetGeometryUserData - returns the user data pointer
  of the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void* rtcGetGeometryUserData(RTCGeometry geometry);
```

DESCRIPTION

The `rtcGetGeometryUserData` function queries the user data pointer previously set with `rtcSetGeometryUserData`. When `rtcSetGeometryUserData` was not called yet, `NULL` is returned.

This function is supposed to be used during rendering, but only supported on the CPU and in SYCL on the GPU.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryUserData

7.64 rtcGetGeometryUserDataFromScene

NAME

```
rtcGetGeometryUserDataFromScene - returns the user data pointer
  of the geometry through the scene object
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void* rtcGetGeometryUserDataFromScene(RTCScene scene, unsigned int geomID);
```

DESCRIPTION

The `rtcGetGeometryUserDataFromScene` function queries the user data pointer previously set with `rtcSetGeometryUserData` from the geometry with index `geomID` from the specified scene `scene`. When `rtcSetGeometryUserData` was not called yet, `NULL` is returned.

This function is supposed to be used during rendering.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryUserData, rtcGetGeometryUserData

7.65 rtcSetGeometryUserPrimitiveCount

NAME

`rtcSetGeometryUserPrimitiveCount - sets the number of primitives of a user-defined geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryUserPrimitiveCount(
  RTCGeometry geometry,
  unsigned int userPrimitiveCount
);
```

DESCRIPTION

The `rtcSetGeometryUserPrimitiveCount` function sets the number of user-defined primitives (`userPrimitiveCount` parameter) of the specified user-defined geometry (`geometry` parameter).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_USER

7.66 rtcSetGeometryBoundsFunction

NAME

```
rtcSetGeometryBoundsFunction - sets a callback to query the
  bounding box of user-defined primitives
```

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCBoundsFunctionArguments
{
  void* geometryUserPtr;
  unsigned int primID;
  unsigned int timeStep;
  struct RTCBounds* bounds_o;
};

typedef void (*RTCBoundsFunction)(
  const struct RTCBoundsFunctionArguments* args
);

void rtcSetGeometryBoundsFunction(
  RTCGeometry geometry,
  RTCBoundsFunction bounds,
  void* userPtr
);
```

DESCRIPTION

The `rtcSetGeometryBoundsFunction` function registers a bounding box callback function (`bounds` argument) with payload (`userPtr` argument) for the specified user geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

In SYCL mode the BVH construction is done on the host and the passed function pointer must be a host-side function pointer.

The registered bounding box callback function is invoked to calculate axis-aligned bounding boxes of the primitives of the user-defined geometry during spatial acceleration structure construction. The bounding box callback of `RTCBoundsFunction` type is invoked with a pointer to a structure of type `RTCBoundsFunctionArguments` which contains various arguments, such as: the user data of the geometry (`geometryUserPtr` member), the ID of the primitive to calculate the bounds for (`primID` member), the time step at which to calculate the bounds (`timeStep` member), and a memory location to write the calculated bound to (`bounds_o` member).

In a typical usage scenario one would store a pointer to the internal representation of the user geometry object using `rtcSetGeometryUserData`. The callback function can then read that pointer from the `geometryUserPtr` field and calculate the proper bounding box for the requested primitive and time, and store that bounding box to the destination structure (`bounds_o` member).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_USER

7.67 rtcSetGeometryIntersectFunction

NAME

```
rtcSetGeometryIntersectFunction - sets the callback function to
  intersect a user geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCIntersectFunctionNArguments
{
  int* valid;
  void* geometryUserPtr;
  unsigned int primID;
  struct RTCRayQueryContext* context;
  struct RTCRayHitN* rayhit;
  unsigned int N;
  unsigned int geomID;
};

typedef void (*RTCIntersectFunctionN)(
  const struct RTCIntersectFunctionNArguments* args
);

void rtcSetGeometryIntersectFunction(
  RTCGeometry geometry,
  RTCIntersectFunctionN intersect
);
```

DESCRIPTION

The `rtcSetGeometryIntersectFunction` function registers a ray/primitive intersection callback function (`intersect` argument) for the specified user geometry (`geometry` argument).

Only a single callback function can be registered per geometry and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered callback function is invoked by `rtcIntersect`-type ray queries to calculate the intersection of a ray packet of variable size with one user-defined primitive. The callback function of type `RTCIntersectFunctionN` gets passed a number of arguments through the `RTCIntersectFunctionNArguments` structure. The value `N` specifies the ray packet size, `valid` points to an array of integers that specify whether the corresponding ray is valid (-1) or invalid (0), the `geometryUserPtr` member points to the geometry user data previously set through `rtcSetGeometryUserData`, the `context` member points to the ray query context passed to the ray query, the `rayhit` member points to a ray and hit packet of variable size `N`, and the `geomID` and `primID` member identifies the geometry ID and primitive ID of the primitive to intersect.

The `ray` component of the `rayhit` structure contains valid data, in particular the `tfar` value is the current closest hit distance found. All data inside the `hit` component of the `rayhit` structure are undefined and should not be read by the function.

The task of the callback function is to intersect each active ray from the ray packet with the specified user primitive. If the user-defined primitive is missed by a ray of the ray packet, the function should return without modifying the ray

or hit. If an intersection of the user-defined primitive with the ray was found in the valid range (from `tnear` to `tfar`), it should update the hit distance of the ray (`tfar` member) and the hit (`u`, `v`, `Ng`, `instID`, `geomID`, `primID` members). In particular, the currently intersected instance is stored in the `instID` field of the ray query context, which must be deep copied into the `instID` member of the hit.

As a primitive might have multiple intersections with a ray, the intersection filter function needs to be invoked by the user geometry intersection callback for each encountered intersection, if filtering of intersections is desired. This can be achieved through the `rtcInvokeIntersectFilterFromGeometry` call.

Within the user geometry intersect function, it is safe to trace new rays and create new scenes and geometries.

When performing ray queries using `rtcIntersect1`, it is guaranteed that the packet size is 1 when the callback is invoked. When performing ray queries using the `rtcIntersect4/8/16` functions, it is not generally guaranteed that the ray packet size (and order of rays inside the packet) passed to the callback matches the initial ray packet. However, under some circumstances these properties are guaranteed, and whether this is the case can be queried using `rtcGetDeviceProperty`.

For many usage scenarios, repacking and re-ordering of rays does not cause difficulties in implementing the callback function. However, algorithms that need to extend the ray with additional data must use the `rayID` component of the ray to identify the original ray to access the per-ray data.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryOccludedFunction, rtcSetGeometryUserData, rtcInvokeIntersectFilterFromGeometry

7.68 rtcSetGeometryOccludedFunction

NAME

```
rtcSetGeometryOccludedFunction - sets the callback function to
  test a user geometry for occlusion
```

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCOccludedFunctionNArguments
{
  int* valid;
  void* geometryUserPtr;
  unsigned int primID;
  struct RTCRayQueryContext* context;
  struct RTCRayN* ray;
  unsigned int N;
  unsigned int geomID;
};

typedef void (*RTCOccludedFunctionN)(
  const struct RTCOccludedFunctionNArguments* args
);

void rtcSetGeometryOccludedFunction(
  RTCGeometry geometry,
  RTCOccludedFunctionN filter
);
```

DESCRIPTION

The `rtcSetGeometryOccludedFunction` function registers a ray/primitive occlusion callback function (`filter` argument) for the specified user geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered callback function is invoked by `rtcOccluded`-type ray queries to test whether the rays of a packet of variable size are occluded by a user-defined primitive. The callback function of type `RTCOccludedFunctionN` gets passed a number of arguments through the `RTCOccludedFunctionNArguments` structure. The value `N` specifies the ray packet size, `valid` points to an array of integers which specify whether the corresponding ray is valid (-1) or invalid (0), the `geometryUserPtr` member points to the geometry user data previously set through `rtcSetGeometryUserData`, the `context` member points to the ray query context passed to the ray query, the `ray` member points to a ray packet of variable size `N`, and the `geomID` and `primID` member identifies the geometry ID and primitive ID of the primitive to intersect.

The task of the callback function is to intersect each active ray from the ray packet with the specified user primitive. If the user-defined primitive is missed by a ray of the ray packet, the function should return without modifying the ray. If an intersection of the user-defined primitive with the ray was found in the valid range (from `tnear` to `tfar`), it should set the `tfar` member of the ray to `-inf`.

As a primitive might have multiple intersections with a ray, the occlusion filter function needs to be invoked by the user geometry occlusion callback for each encountered intersection, if filtering of intersections is desired. This can be achieved through the `rtcInvokeOccludedFilterFromGeometry` call.

Within the user geometry occlusion function, it is safe to trace new rays and create new scenes and geometries.

When performing ray queries using `rtcOccluded1`, it is guaranteed that the packet size is 1 when the callback is invoked. When performing ray queries using the `rtcOccluded4/8/16` functions, it is not generally guaranteed that the ray packet size (and order of rays inside the packet) passed to the callback matches the initial ray packet. However, under some circumstances these properties are guaranteed, and whether this is the case can be queried using `rtcGetDeviceProperty`.

For many usage scenarios, repacking and re-ordering of rays does not cause difficulties in implementing the callback function. However, algorithms that need to extend the ray with additional data must use the `rayID` component of the ray to identify the original ray to access the per-ray data.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryIntersectFunction, rtcSetGeometryUserData, rtcInvokeOccludedFilterFromGeometry

7.69 rtcSetGeometryPointQueryFunction

NAME

```
rtcSetGeometryPointQueryFunction - sets the point query callback function
  for a geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCPointQueryFunctionArguments
{
  // the (world space) query object that was passed as an argument of rtcPointQuery.
  struct RTCPointQuery* query;

  // used for user input/output data. Will not be read or modified internally.
  void* userPtr;

  // primitive and geometry ID of primitive
  unsigned int  primID;
  unsigned int  geomID;

  // the context with transformation and instance ID stack
  struct RTCPointQueryContext* context;

  // scaling factor indicating whether the current instance transformation
  // is a similarity transformation.
  float similarityScale;
};

typedef bool (*RTCPointQueryFunction)(
  struct RTCPointQueryFunctionArguments* args
);

void rtcSetGeometryPointQueryFunction(
  RTCGeometry geometry,
  RTCPointQueryFunction queryFunc
);
```

DESCRIPTION

The `rtcSetGeometryPointQueryFunction` function registers a point query callback function (`queryFunc` argument) for the specified geometry (`geometry` argument).

Only a single callback function can be registered per geometry and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered callback function is invoked by rtcPointQuery for every primitive of the geometry that intersects the corresponding point query domain. The callback function of type `RTCPointQueryFunction` gets passed a number of arguments through the `RTCPointQueryFunctionArguments` structure. The `query` object is the original point query object passed into rtcPointQuery, `userPtr` is an arbitrary pointer to pass input into and store results of the callback function. The `primID`, `geomID` and `context` (see rtcInitPointQueryContext for details) can be used to identify the geometry data of the primitive.

A `RTCPointQueryFunction` can also be passed directly as an argument to rtcPointQuery. In this case the callback is invoked for all primitives in the scene that intersect the query domain. If a callback function is passed as an argument to rtcPointQuery and (a potentially different) callback function is set for a geometry with rtcSetGeometryPointQueryFunction both callback functions are invoked and the callback function passed to rtcPointQuery will be called before the geometry specific callback function.

If instancing is used, the parameter `simliarityScale` indicates whether the current instance transform (top element of the stack in `context`) is a similarity transformation or not. Similarity transformations are composed of translation, rotation and uniform scaling and if a matrix M defines a similarity transformation, there is a scaling factor D such that for all x,y: dist(Mx, My) = D * dist(x, y). In this case the parameter `scalingFactor` is this scaling factor D and otherwise it is 0. A valid similarity scale (`similarityScale` > 0) allows to compute distance information in instance space and scale the distances into world space (for example, to update the query radius, see below) by dividing the instance space distance with the similarity scale. If the current instance transform is not a similarity transform (`similarityScale` is 0), the distance computation has to be performed in world space to ensure correctness. In this case the instance to world transformations given with the `context` should be used to transform the primitive data into world space. Otherwise, the query location can be transformed into instance space which can be more efficient. If there is no instance transform, the similarity scale is 1.

The callback function will potentially be called for primitives outside the query domain for two reasons: First, the callback is invoked for all primitives inside a BVH leaf node since no geometry data of primitives is determined internally and therefore individual primitives are not culled (only their (aggregated) bounding boxes). Second, in case non similarity transformations are used, the resulting ellipsoidal query domain (in instance space) is approximated by its axis aligned bounding box internally and therefore inner nodes that do not intersect the original domain might intersect the approximative bounding box which results in unnecessary callbacks. In any case, the callbacks are conservative, i.e. if a primitive is inside the query domain a callback will be invoked but the reverse is not necessarily true.

For efficiency, the radius of the `query` object can be decreased (in world space) inside the callback function to improve culling of geometry during BVH traversal. If the query radius was updated, the callback function should return `true` to issue an update of internal traversal information. Increasing the radius or modifying the time or position of the query results in undefined behaviour.

Within the callback function, it is safe to call rtcPointQuery again, for example when implementing instancing manually. In this case the instance transformation should be pushed onto the stack in `context`. Embree will internally compute the point query information in instance space using the top element of the stack in `context` when rtcPointQuery is called.

For a reference implementation of a closest point traversal of triangle meshes using instancing and user defined instancing see the tutorial [ClosestPoint].

SEE ALSO

rtcPointQuery, rtcInitPointQueryContext

7.70 rtcGetSYCLDeviceFunctionPointer

NAME

```
rtcGetSYCLDeviceFunctionPointer - obtains a device side
  function pointer for some SYCL function
```

SYNOPSIS

```
#include <embree4/rtcore.h>

template<auto F>
inline decltype(F) rtcGetSYCLDeviceFunctionPointer(sycl::queue& queue);
```

DESCRIPTION

This function returns a device side function pointer for some function F. This function F must be defined using the `RTC_SYCL_INDIRECTLY_CALLABLE` attribute, e.g.:

```
RTC_SYCL_INDIRECTLY_CALLABLE void filter(
  const RTCFilterFunctionNArguments* args) { ... }

RTCFilterFunctionN fptr = rtcGetSYCLDeviceFunctionPointer<filter>(queue);
```

Such a device side function pointers of some filter callbacks can get assigned to a geometry using the `rtcSetGeometryIntersectFilterFunction` and `rtcSetGeometryOccludedFilterFunction` API functions.

Further, device side function pointers for user geometry callbacks can be assigned to geometries using the `rtcSetGeometryIntersectFunction` and `rtcSetGeometryOccludedFunction` API calls.

These geometry versions of the callback functions are disabled in SYCL by default, and we recommend not using them for performance reasons.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryIntersectFunction, rtcSetGeometryOccludedFunction, rtcSetGeometryIntersectFilterFunction, rtcSetGeometryOccludedFilterFunction

7.71 rtcSetGeometryInstancedScene

NAME

```
rtcSetGeometryInstancedScene - sets the instanced scene of
  an instance geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryInstancedScene(
  RTCGeometry geometry,
  RTCScene scene
);
```

DESCRIPTION

The `rtcSetGeometryInstancedScene` function sets the instanced scene (`scene` argument) of the specified instance geometry (`geometry` argument).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_INSTANCE, rtcSetGeometryTransform

7.72 rtcSetGeometryTransform

NAME

```
rtcSetGeometryTransform - sets the transformation for a particular
  time step of an instance geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryTransform(
  RTCGeometry geometry,
  unsigned int timeStep,
  enum RTCFormat format,
  const float* xfm
);
```

DESCRIPTION

The `rtcSetGeometryTransform` function sets the local-to-world affine transformation (`xfm` parameter) of an instance geometry (`geometry` parameter) for a particular time step (`timeStep` parameter). The transformation is specified as a 3×4 matrix (3×3 linear transformation plus translation), for which the following formats (`format` parameter) are supported:

- `RTC_FORMAT_FLOAT3X4_ROW_MAJOR`: The 3×4 float matrix is laid out in row-major form.
- `RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form.
- `RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form as a 4×4 homogeneous matrix with the last row being equal to (0, 0, 0, 1).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_INSTANCE

7.73 rtcSetGeometryTransformQuaternion

NAME

```
rtcSetGeometryTransformQuaternion - sets the transformation for a particular
  time step of an instance geometry as a decomposition of the
  transformation matrix using quaternions to represent the rotation.
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryTransformQuaternion(
  RTCGeometry geometry,
  unsigned int timeStep,
  const struct RTCQuaternionDecomposition* qd
);
```

DESCRIPTION

The `rtcSetGeometryTransformQuaternion` function sets the local-to-world affine transformation (`qd` parameter) of an instance geometry (`geometry` parameter) for a particular time step (`timeStep` parameter). The transformation is specified as a RTCQuaternionDecomposition, which is a decomposition of an affine transformation that represents the rotational component of an affine transformation as a quaternion. This allows interpolating rotational transformations exactly using spherical linear interpolation (such as a turning wheel).

For more information about the decomposition see RTCQuaternionDecomposition. The quaternion given in the `RTCQuaternionDecomposition` struct will be normalized internally.

For correct results, the transformation matrices for all time steps must be set either using `rtcSetGeometryTransform` or `rtcSetGeometryTransformQuaternion`. Mixing both representations is not allowed. Spherical linear interpolation will be used, iff the transformation matizes are set with `rtcSetGeometryTransformQuaternion`.

For an example of this feature see the tutorial Quaternion Motion Blur.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcInitQuaternionDecomposition, rtcSetGeometryTransform

7.74 rtcGetGeometryTransform

NAME

```
rtcGetGeometryTransform - returns the interpolated instance
  transformation for the specified time
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcGetGeometryTransform(
  RTCGeometry geometry,
  float time,
  enum RTCFormat format,
  void* xfm
);
```

DESCRIPTION

The `rtcGetGeometryTransform` function returns the interpolated local to world transformation (`xfm` parameter) of an instance geometry (`geometry` parameter) for a particular time (`time` parameter in range $[0, 1]$) in the specified format (`format` parameter).

Possible formats for the returned matrix are:

- `RTC_FORMAT_FLOAT3X4_ROW_MAJOR`: The 3×4 float matrix is laid out in row-major form.
- `RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form.
- `RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form as a 4×4 homogeneous matrix with last row equal to (0, 0, 0, 1).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_INSTANCE, rtcSetGeometryTransform

7.75 rtcSetGeometryTessellationRate

NAME

`rtcSetGeometryTessellationRate - sets the tessellation rate of the geometry`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryTessellationRate(
  RTCGeometry geometry,
  float tessellationRate
);
```

DESCRIPTION

The `rtcSetGeometryTessellationRate` function sets the tessellation rate (`tessellationRate` argument) for the specified geometry (`geometry` argument). The tessellation rate can only be set for flat curves and subdivision geometries. For curves, the tessellation rate specifies the number of ray-facing quads per curve segment. For subdivision surfaces, the tessellation rate specifies the number of quads along each edge.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_CURVE, RTC_GEOMETRY_TYPE_SUBDIVISION

7.76 rtcSetGeometryTopologyCount

NAME

```
rtcSetGeometryTopologyCount - sets the number of topologies of
  a subdivision geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryTopologyCount(
  RTCGeometry geometry,
  unsigned int topologyCount
);
```

DESCRIPTION

The `rtcSetGeometryTopologyCount` function sets the number of topologies (`topologyCount` parameter) for the specified subdivision geometry (`geometry` parameter). The number of topologies of a subdivision geometry must be greater or equal to 1.

To use multiple topologies, first the number of topologies must be specified, then the individual topologies can be configured using `rtcSetGeometrySubdivisionMode` and by setting an index buffer (`RTC_BUFFER_TYPE_INDEX`) using the topology ID as the buffer slot.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_SUBDIVISION, rtcSetGeometrySubdivisionMode

7.77 rtcSetGeometrySubdivisionMode

NAME

```
rtcSetGeometrySubdivisionMode - sets the subdivision mode
  of a subdivision geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometrySubdivisionMode(
  RTCGeometry geometry,
  unsigned int topologyID,
  enum RTCSubdivisionMode mode
);
```

DESCRIPTION

The `rtcSetGeometrySubdivisionMode` function sets the subdivision mode (`mode` parameter) for the topology (`topologyID` parameter) of the specified subdivision geometry (`geometry` parameter).

The subdivision modes can be used to force linear interpolation for certain parts of the subdivision mesh:

- `RTC_SUBDIVISION_MODE_NO_BOUNDARY`: Boundary patches are ignored. This way each rendered patch has a full set of control vertices.
- `RTC_SUBDIVISION_MODE_SMOOTH_BOUNDARY`: The sequence of boundary control points are used to generate a smooth B-spline boundary curve (default mode).
- `RTC_SUBDIVISION_MODE_PIN_CORNERS`: Corner vertices are pinned to their location during subdivision.
- `RTC_SUBDIVISION_MODE_PIN_BOUNDARY`: All vertices at the border are pinned to their location during subdivision. This way the boundary is interpolated linearly. This mode is typically used for texturing to also map texels at the border of the texture to the mesh.
- `RTC_SUBDIVISION_MODE_PIN_ALL`: All vertices at the border are pinned to their location during subdivision. This way all patches are linearly interpolated.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_SUBDIVISION

7.78 rtcSetGeometryVertexAttributeTopology

NAME

```
rtcSetGeometryVertexAttributeTopology - binds a vertex
  attribute to a topology of the geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcSetGeometryVertexAttributeTopology(
  RTCGeometry geometry,
  unsigned int vertexAttributeID,
  unsigned int topologyID
);
```

DESCRIPTION

The `rtcSetGeometryVertexAttributeTopology` function binds a vertex attribute buffer slot (`vertexAttributeID` argument) to a topology (`topologyID` argument) for the specified subdivision geometry (`geometry` argument). Standard vertex buffers are always bound to the default topology (topology 0) and cannot be bound differently. A vertex attribute buffer always uses the topology it is bound to when used in the `rtcInterpolate` and `rtcInterpolateN` calls.

A topology with ID `i` consists of a subdivision mode set through `rtcSetGeometrySubdivisionMode` and the index buffer bound to the index buffer slot `i`. This index buffer can assign indices for each face of the subdivision geometry that are different to the indices of the default topology. These new indices can for example be used to introduce additional borders into the subdivision mesh to map multiple textures onto one subdivision geometry.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometrySubdivisionMode, rtcInterpolate, rtcInterpolateN

7.79 rtcSetGeometryDisplacementFunction

NAME

```
rtcSetGeometryDisplacementFunction - sets the displacement function
  for a subdivision geometry
```

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCDisplacementFunctionNArguments
{
  void* geometryUserPtr;
  RTCGeometry geometry;
  unsigned int primID;
  unsigned int timeStep;
  const float* u;
  const float* v;
  const float* Ng_x;
  const float* Ng_y;
  const float* Ng_z;
  float* P_x;
  float* P_y;
  float* P_z;
  unsigned int N;
};

typedef void (*RTCDisplacementFunctionN)(
   const struct RTCDisplacementFunctionNArguments* args
);

void rtcSetGeometryDisplacementFunction(
  RTCGeometry geometry,
  RTCDisplacementFunctionN displacement
);
```

DESCRIPTION

The `rtcSetGeometryDisplacementFunction` function registers a displacement callback function (`displacement` argument) for the specified subdivision geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered displacement callback function is invoked to displace points on the subdivision geometry during spatial acceleration structure construction, during the `rtcCommitScene` call.

The callback function of type `RTCDisplacementFunctionN` is invoked with a number of arguments stored inside the `RTCDisplacementFunctionNArguments` structure. The provided user data pointer of the geometry (`geometryUserPtr` member) can be used to point to the application's representation of the subdivision mesh. A number `N` of points to displace are specified in a structure of array layout. For each point to displace, the local patch UV coordinates (`u` and `v` arrays), the normalized geometry normal (`Ng_x`, `Ng_y`, and `Ng_z` arrays), and the position (`P_x`, `P_y`, and `P_z` arrays) are provided. The task of the displacement function is to use this information and change the position data.

The geometry handle (`geometry` member) and primitive ID (`primID` member) of the patch to displace are additionally provided as well as the time step `timeStep`, which can be important if the displacement is time-dependent and motion blur is used.

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

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

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_SUBDIVISION

7.80 rtcGetGeometryFirstHalfEdge

NAME

`rtcGetGeometryFirstHalfEdge - returns the first half edge of a face`

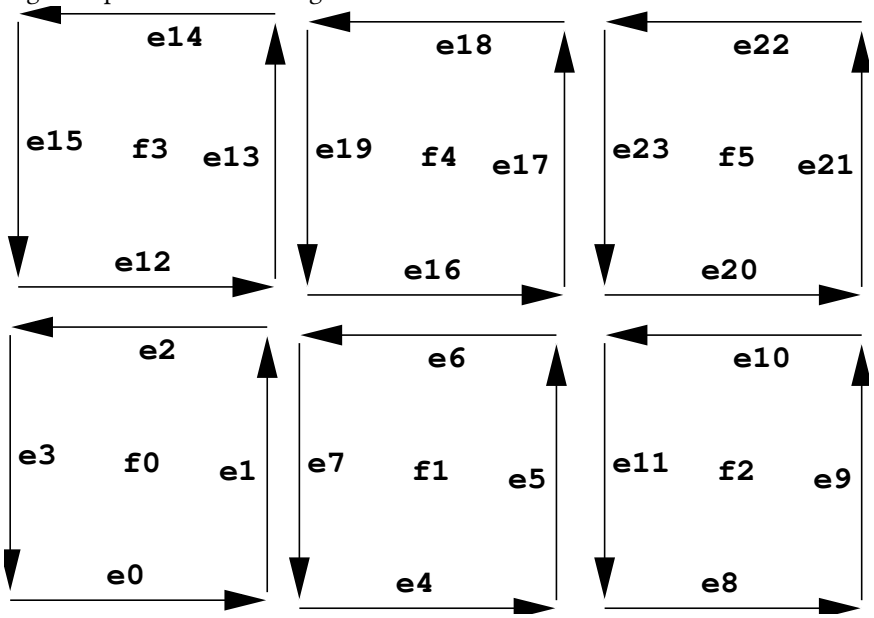
SYNOPSIS

```
#include <embree4/rtcore.h>

unsigned int rtcGetGeometryFirstHalfEdge(
  RTCGeometry geometry,
  unsigned int faceID
);
```

DESCRIPTION

The `rtcGetGeometryFirstHalfEdge` function returns the ID of the first half edge belonging to the specified face (`faceID` argument). For instance in the following example the first half edge of face `f1` is `e4`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

Here f0 to f7 are 8 quadrilateral faces with 4 vertices each. The edges e0 to e23 of these faces are shown with their orientation. For each face the ID of the edges corresponds to the slots the face occupies in the index array of the geometry. E.g. as the indices of face f1 start at location 4 of the index array, the first edge is edge e4, the next edge e5, etc.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, rtcGetGeometryFace, rtcGetGeometryOppositeHalfEdge, rtcGetGeometryNextHalfEdge, rtcGetGeometryPreviousHalfEdge

7.81 rtcGetGeometryFace

NAME

`rtcGetGeometryFace - returns the face of some half edge`

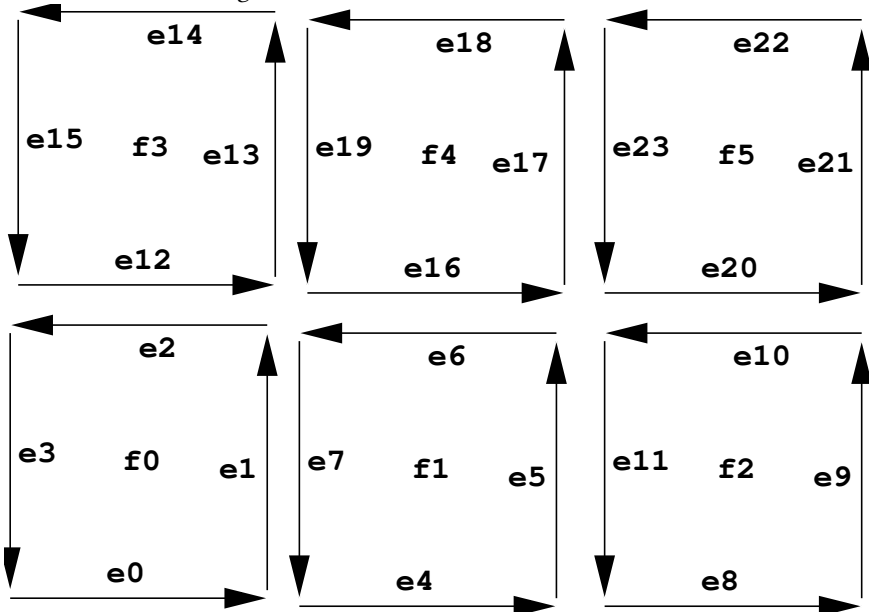
SYNOPSIS

```
#include <embree4/rtcore.h>

unsigned int rtcGetGeometryFace(
  RTCGeometry geometry,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryFace` function returns the ID of the face the specified half edge (`edgeID` argument) belongs to. For instance in the following example the face `f1` is returned for edges `e4`, `e5`, `e6`, and `e7`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, rtcGetGeometryFace, rtcGetGeometryOppositeHalfEdge, rtcGetGeometryNextHalfEdge, rtcGetGeometryPreviousHalfEdge

7.82 rtcGetGeometryNextHalfEdge

NAME

`rtcGetGeometryNextHalfEdge - returns the next half edge`

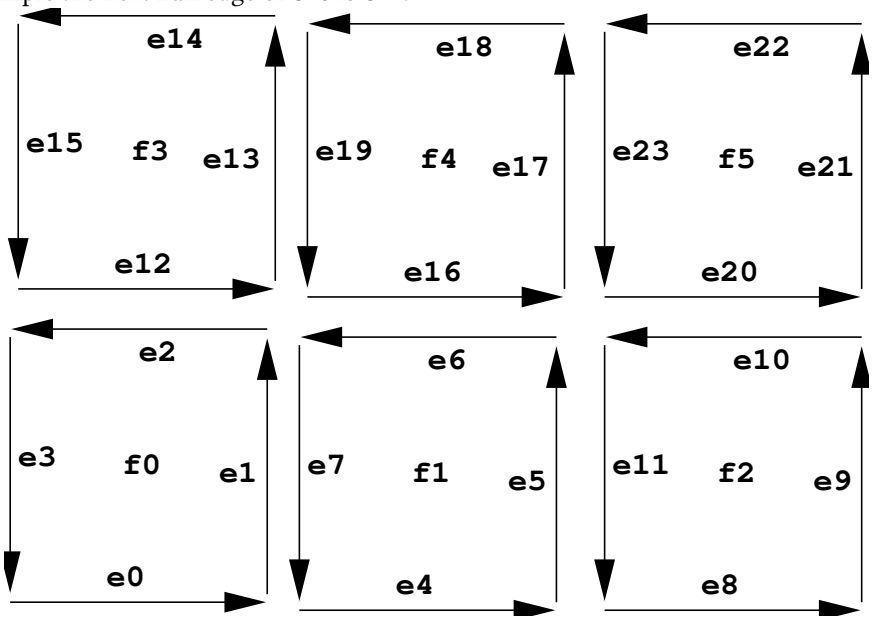
SYNOPSIS

```
#include <embree4/rtcore.h>

unsigned int rtcGetGeometryNextHalfEdge(
  RTCGeometry geometry,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryNextHalfEdge` function returns the ID of the next half edge of the specified half edge (`edgeID` argument). For instance in the following example the next half edge of `e10` is `e11`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, rtcGetGeometryFace, rtcGetGeometryOppositeHalfEdge, rtcGetGeometryNextHalfEdge, rtcGetGeometryPreviousHalfEdge

7.83 rtcGetGeometryPreviousHalfEdge

NAME

`rtcGetGeometryPreviousHalfEdge - returns the previous half edge`

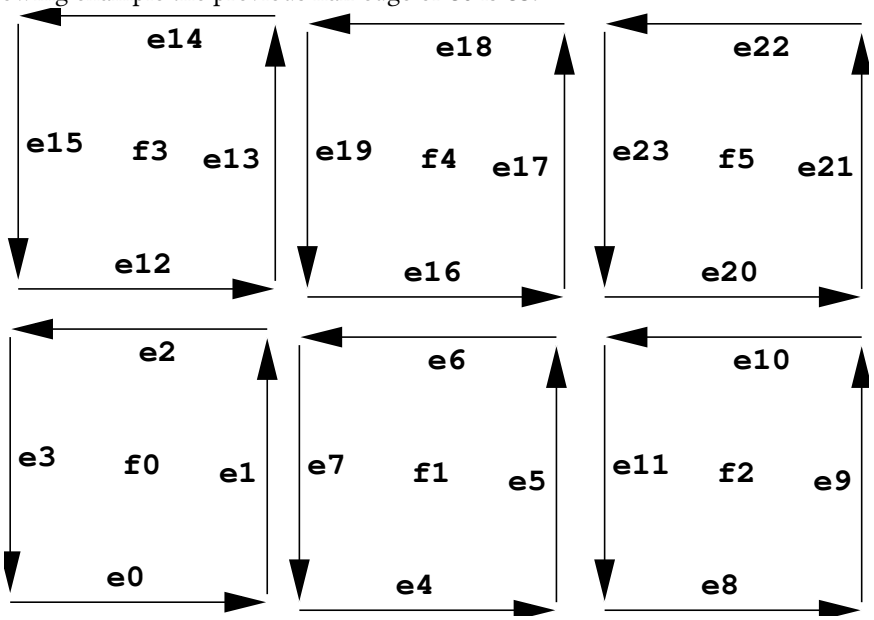
SYNOPSIS

```
#include <embree4/rtcore.h>

unsigned int rtcGetGeometryPreviousHalfEdge(
  RTCGeometry geometry,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryPreviousHalfEdge` function returns the ID of the previous half edge of the specified half edge (`edgeID` argument). For instance in the following example the previous half edge of `e6` is `e5`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, rtcGetGeometryFace, rtcGetGeometryOppositeHalfEdge, rtcGetGeometryNextHalfEdge, rtcGetGeometryPreviousHalfEdge

7.84 rtcGetGeometryOppositeHalfEdge

NAME

`rtcGetGeometryOppositeHalfEdge - returns the opposite half edge`

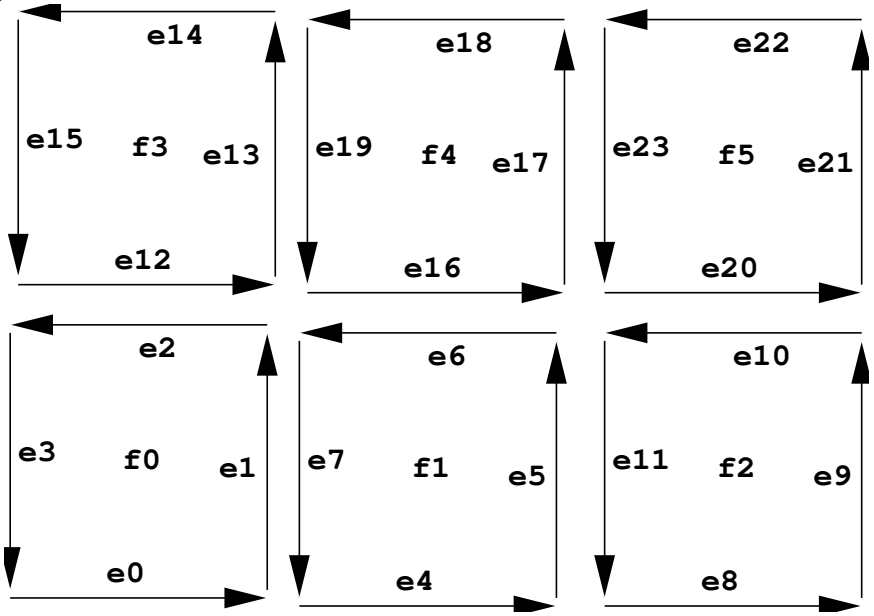
SYNOPSIS

```
#include <embree4/rtcore.h>

unsigned int rtcGetGeometryOppositeHalfEdge(
  RTCGeometry geometry,
  unsigned int topologyID,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryOppositeHalfEdge` function returns the ID of the opposite half edge of the specified half edge (`edgeID` argument) in the specified topology (`topologyID` argument). For instance in the following example the opposite half edge of `e6` is `e16`.



An opposite half edge does not exist if the specified half edge has either no neighboring face, or more than 2 neighboring faces. In these cases the function just returns the same edge `edgeID` again.

This function can only be used for subdivision geometries. The function depends on the topology as the topologies of a subdivision geometry have different index buffers assigned.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, rtcGetGeometryFace, rtcGetGeometryOppositeHalfEdge, rtcGetGeometryNextHalfEdge, rtcGetGeometryPreviousHalfEdge

7.85 rtcInterpolate

NAME

rtcInterpolate - interpolates vertex attributes

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCInterpolateArguments
{
  RTCGeometry geometry;
  unsigned int primID;
  float u;
  float v;
  enum RTCBufferType bufferType;
  unsigned int bufferSlot;
  float* P;
  float* dPdu;
  float* dPdv;
  float* ddPdudu;
  float* ddPdvdv;
  float* ddPdudv;
  unsigned int valueCount;
};

void rtcInterpolate(
  const struct RTCInterpolateArguments* args
);
```

DESCRIPTION

The `rtcInterpolate` function smoothly interpolates per-vertex data over the geometry. This interpolation is supported for triangle meshes, quad meshes, curve geometries, and subdivision geometries. Apart from interpolating the vertex attribute itself, it is also possible to get the first and second order derivatives of that value. This interpolation ignores displacements of subdivision surfaces and always interpolates the underlying base surface.

The `rtcInterpolate` call gets passed a number of arguments inside a structure of type `RTCInterpolateArguments`. For some geometry (`geometry` parameter) this function smoothly interpolates the per-vertex data stored inside the specified geometry buffer (`bufferType` and `bufferSlot` parameters) to the u/v location (`u` and `v` parameters) of the primitive (`primID` parameter). The number of floating point values to interpolate and store to the destination arrays can be specified using the `valueCount` parameter. As interpolation buffer, one can specify vertex buffers (`RTC_BUFFER_TYPE_VERTEX`) and vertex attribute buffers (`RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`) as well.

The `rtcInterpolate` call stores `valueCount` number of 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`.

To use `rtcInterpolate` for a geometry, all changes to that geometry must be properly committed using `rtcCommitGeometry`.

All input buffers and output arrays must be padded to 16 bytes, as the implementation uses 16-byte SSE instructions to read and write into these buffers.

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

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcInterpolateN

7.86 rtcInterpolateN

NAME

`rtcInterpolateN - performs N interpolations of vertex attribute data`

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCInterpolateNArguments
{
  RTCGeometry geometry;
  const void* valid;
  const unsigned int* primIDs;
  const float* u;
  const float* v;
  unsigned int N;
  enum RTCBufferType bufferType;
  unsigned int bufferSlot;
  float* P;
  float* dPdu;
  float* dPdv;
  float* ddPdudu;
  float* ddPdvdv;
  float* ddPdudv;
  unsigned int valueCount;
};

void rtcInterpolateN(
  const struct RTCInterpolateNArguments* args
);
```

DESCRIPTION

The `rtcInterpolateN` is similar to `rtcInterpolate`, but performs `N` many interpolations at once. It additionally gets an array of u/v coordinates and a valid mask (`valid` parameter) that specifies which of these coordinates are valid. The valid mask points to `N` 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. The value `N` must be divisible by 4.

To use `rtcInterpolateN` for a geometry, all changes to that geometry must be properly committed using `rtcCommitGeometry`.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcInterpolate

7.87 rtcNewBuffer

NAME

`rtcNewBuffer - creates a new data buffer`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCBuffer rtcNewBuffer(
  RTCDevice device,
  size_t byteSize
);
```

DESCRIPTION

The `rtcNewBuffer` function creates a new data buffer object of specified size in bytes (`byteSize` argument) that is bound to the specified device (`device` argument). The buffer object is reference counted with an initial reference count of 1. The returned buffer object can be released using the `rtcReleaseBuffer` API call. The specified number of bytes are allocated at buffer construction time and deallocated when the buffer is destroyed.

When the buffer will be used as a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` and `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`), the last buffer element must be readable using 16-byte SSE load instructions, thus padding the last element is required for certain layouts. E.g. a standard `float3` vertex buffer layout should add storage for at least one more float to the end of the buffer.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainBuffer, rtcReleaseBuffer

7.88 rtcNewSharedBuffer

NAME

`rtcNewSharedBuffer - creates a new shared data buffer`

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCBuffer rtcNewSharedBuffer(
  RTCDevice device,
  void* ptr,
  size_t byteSize
);
```

DESCRIPTION

The `rtcNewSharedBuffer` function creates a new shared data buffer object bound to the specified device (`device` argument). The buffer object is reference counted with an initial reference count of 1. The buffer can be released using the `rtcReleaseBuffer` function.

At construction time, the pointer to the user-managed buffer data (`ptr` argument) including its size in bytes (`byteSize` argument) is provided to create the buffer. At buffer construction time no buffer data is allocated, but the buffer data provided by the application is used. The buffer data must remain valid for as long as the buffer may be used, and the user is responsible to free the buffer data when no longer required.

When the buffer will be used as a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` and `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`), the last buffer element must be readable using 16-byte SSE load instructions, thus padding the last element is required for certain layouts. E.g. a standard `float3` vertex buffer layout should add storage for at least one more float to the end of the buffer.

The data pointer (`ptr` argument) must be aligned to 4 bytes; otherwise the `rtcNewSharedBuffer` function will fail.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainBuffer, rtcReleaseBuffer

7.89 rtcRetainBuffer

NAME

`rtcRetainBuffer - increments the buffer reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcRetainBuffer(RTCBuffer buffer);
```

DESCRIPTION

Buffer objects are reference counted. The `rtcRetainBuffer` function increments the reference count of the passed buffer object (`buffer` argument). This function together with `rtcReleaseBuffer` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBuffer, rtcReleaseBuffer

7.90 rtcReleaseBuffer

NAME

`rtcReleaseBuffer - decrements the buffer reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcReleaseBuffer(RTCBuffer buffer);
```

DESCRIPTION

Buffer objects are reference counted. The `rtcReleaseBuffer` function decrements the reference count of the passed buffer object (`buffer` argument). When the reference count falls to 0, the buffer gets destroyed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBuffer, rtcRetainBuffer

7.91 rtcGetBufferData

NAME

`rtcGetBufferData - gets a pointer to the buffer data`

SYNOPSIS

```
#include <embree4/rtcore.h>

void* rtcGetBufferData(RTCBuffer buffer);
```

DESCRIPTION

The `rtcGetBufferData` function returns a pointer to the buffer data of the specified buffer object (`buffer` argument).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBuffer

7.92 RTCRay

NAME

RTCRay - single ray structure

SYNOPSIS

```
#include <embree4/rtcore_ray.h>

struct RTC_ALIGN(16) RTCRay
{
  float org_x;        // x coordinate of ray origin
  float org_y;        // y coordinate of ray origin
  float org_z;        // z coordinate of ray origin
  float tnear;        // start of ray segment

  float dir_x;        // x coordinate of ray direction
  float dir_y;        // y coordinate of ray direction
  float dir_z;        // z coordinate of ray direction
  float time;         // time of this ray for motion blur

  float tfar;         // end of ray segment (set to hit distance)
  unsigned int mask;  // ray mask
  unsigned int id;    // ray ID
  unsigned int flags; // ray flags
};
```

DESCRIPTION

The RTCRay structure defines the ray layout for a single ray. The ray contains the origin (org_x, org_y, org_z members), direction vector (dir_x, dir_y, dir_z members), and ray segment (tnear and tfar members). The ray direction does not have to be normalized, and only the parameter range specified by the tnear/tfar interval is considered valid.

The ray segment must be in the range $[0, \infty]$, thus ranges that start behind the ray origin are not allowed, but ranges can reach to infinity.

The ray further contains a motion blur time in the range $[0, 1]$ (time member), a ray mask (mask member), a ray ID (id member), and ray flags (flags member). The ray mask can be used to mask out some geometries for some rays (see rtcSetGeometryMask for more details). The ray ID can be used to identify a ray inside a callback function, even if the order of rays inside a ray packet has changed.

The embree4/rtcore_ray.h header additionally defines the same ray structure in structure of array (SOA) layout for API functions accepting ray packets of size 4 (RTCRay4 type), size 8 (RTCRay8 type), and size 16 (RTCRay16 type). The header additionally defines an RTCRayNt template for ray packets of an arbitrary compile-time size.

EXIT STATUS

SEE ALSO

RTCHit

7.93 RTCHit

NAME

RTCHit - single hit structure

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCHit
{
  float Ng_x;                                     // x coordinate of geometry normal
  float Ng_y;                                     // y coordinate of geometry normal
  float Ng_z;                                     // z coordinate of geometry normal

  float u;                                        // barycentric u coordinate of hit
  float v;                                        // barycentric v coordinate of hit

  unsigned int primID;                            // geometry ID
  unsigned int geomID;                            // primitive ID
  unsigned int instID[RTC_MAX_INSTANCE_LEVEL_COUNT]; // instance ID
};
```

DESCRIPTION

The RTCHit type defines the type of a ray/primitive intersection result. The hit contains the unnormalized geometric normal in object space at the hit location (Ng_x, Ng_y, Ng_z members), the barycentric u/v coordinates of the hit (u and v members), as well as the primitive ID (primID member), geometry ID (geomID member), and instance ID stack (instID member) of the hit. The parametric intersection distance is not stored inside the hit, but stored inside the tfar member of the ray.

The embree4/rtcore_ray.h header additionally defines the same hit structure in structure of array (SOA) layout for hit packets of size 4 (RTCHit4 type), size 8 (RTCHit8 type), and size 16 (RTCHit16 type). The header additionally defines an RTCHitNt template for hit packets of an arbitrary compile-time size.

EXIT STATUS

SEE ALSO

RTCRay, [Multi-Level Instancing]

7.94 RTCRayHit

NAME

`RTCRayHit - combined single ray/hit structure`

SYNOPSIS

```
#include <embree4/rtcore_ray.h>

struct RTCORE_ALIGN(16) RTCRayHit
{
  struct RTCRay ray;
  struct RTCHit hit;
};
```

DESCRIPTION

The `RTCRayHit` structure is used as input for the `rtcIntersect`-type functions and stores the ray to intersect and some hit fields that hold the intersection result afterwards.

The `embree4/rtcore_ray.h` header additionally defines the same ray/hit structure in structure of array (SOA) layout for API functions accepting ray packets of size 4 (`RTCRayHit4` type), size 8 (`RTCRayHit8` type), and size 16 (`RTCRayHit16` type). The header additionally defines an `RTCRayHitNt` template to generate ray/hit packets of an arbitrary compile-time size.

EXIT STATUS

SEE ALSO

RTCRay, RTCHit

7.95 RTCRayN

NAME

RTCRayN - ray packet of runtime size

SYNOPSIS

```
#include <embree4/rtcore_ray.h>

struct RTCRayN;

float& RTCRayN_org_x(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_org_y(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_org_z(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_tnear(RTCRayN* ray, unsigned int N, unsigned int i);

float& RTCRayN_dir_x(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_dir_y(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_dir_z(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_time (RTCRayN* ray, unsigned int N, unsigned int i);

float&        RTCRayN_tfar (RTCRayN* ray, unsigned int N, unsigned int i);
unsigned int& RTCRayN_mask (RTCRayN* ray, unsigned int N, unsigned int i);
unsigned int& RTCRayN_id   (RTCRayN* ray, unsigned int N, unsigned int i);
unsigned int& RTCRayN_flags(RTCRayN* ray, unsigned int N, unsigned int i);
```

DESCRIPTION

When the ray packet size is not known at compile time (e.g. when Embree returns a ray packet in the `RTCFilterFuncN` callback function), Embree uses the `RTCRayN` type for ray packets. These ray packets can only have sizes of 1, 4, 8, or 16. No other packet size will be used.

You can either implement different special code paths for each of these possible packet sizes and cast the ray to the appropriate ray packet type, or implement one general code path that uses the `RTCRayN_XXX` helper functions to access the ray packet components.

These helper functions get a pointer to the ray packet (`ray` argument), the packet size (`N` argument), and returns a reference to a component (e.g. x-component of origin) of the the i-th ray of the packet (`i` argument).

EXIT STATUS

SEE ALSO

RTCHitN

7.96 RTCHitN

NAME

RTCHitN - hit packet of runtime size

SYNOPSIS

```
#include <embree4/rtcore.h>

struct HitN;

float& RTCHitN_Ng_x(RTCHitN* hit, unsigned int N, unsigned int i);
float& RTCHitN_Ng_y(RTCHitN* hit, unsigned int N, unsigned int i);
float& RTCHitN_Ng_z(RTCHitN* hit, unsigned int N, unsigned int i);

float& RTCHitN_u(RTCHitN* hit, unsigned int N, unsigned int i);
float& RTCHitN_v(RTCHitN* hit, unsigned int N, unsigned int i);

unsigned& RTCHitN_primID(RTCHitN* hit, unsigned int N, unsigned int i);
unsigned& RTCHitN_geomID(RTCHitN* hit, unsigned int N, unsigned int i);
unsigned& RTCHitN_instID(RTCHitN* hit, unsigned int N, unsigned int i, unsigned int level);
```

DESCRIPTION

When the hit packet size is not known at compile time (e.g. when Embree returns a hit packet in the `RTCFilterFuncN` callback function), Embree uses the `RTCHitN` type for hit packets. These hit packets can only have sizes of 1, 4, 8, or 16. No other packet size will be used.

You can either implement different special code paths for each of these possible packet sizes and cast the hit to the appropriate hit packet type, or implement one general code path that uses the `RTCHitN_XXX` helper functions to access hit packet components.

These helper functions get a pointer to the hit packet (`hit` argument), the packet size (`N` argument), and returns a reference to a component (e.g. x component of `Ng`) of the the i-th hit of the packet (`i` argument).

EXIT STATUS

SEE ALSO

RTCRayN

7.97 RTCRayHitN

NAME

RTCRayHitN - combined ray/hit packet of runtime size

SYNOPSIS

```
#include <embree4/rtcore_ray.h>

struct RTCRayHitN;

struct RTCRayN* RTCRayHitN_RayN(struct RTCRayHitN* rayhit, unsigned int N);
struct RTCHitN* RTCRayHitN_HitN(struct RTCRayHitN* rayhit, unsigned int N);
```

DESCRIPTION

When the packet size of a ray/hit structure is not known at compile time (e.g. when Embree returns a ray/hit packet in the RTCIntersectFunctionN callback function), Embree uses the RTCRayHitN type for ray packets. These ray/hit packets can only have sizes of 1, 4, 8, or 16. No other packet size will be used.

You can either implement different special code paths for each of these possible packet sizes and cast the ray/hit to the appropriate ray/hit packet type, or extract the RTCRayN and RTCHitN components using the rtcGetRayN and rtcGetHitN helper functions and use the RTCRayN_XXX and RTCHitN_XXX functions to access the ray and hit parts of the structure.

EXIT STATUS

SEE ALSO

RTCHitN

7.98 RTCFeatureFlags

NAME

RTCFeatureFlags - specifies features to enable
  for ray queries

SYNOPSIS

```
#include <embree4/rtcore_ray.h>

enum RTCFeatureFlags
{
  RTC_FEATURE_FLAG_NONE = 0,

  RTC_FEATURE_FLAG_MOTION_BLUR = 1 << 0,

  RTC_FEATURE_FLAG_TRIANGLE = 1 << 1,
  RTC_FEATURE_FLAG_QUAD = 1 << 2,
  RTC_FEATURE_FLAG_GRID = 1 << 3,
  RTC_FEATURE_FLAG_SUBDIVISION = 1 << 4,
  RTC_FEATURE_FLAG_POINT = ... ,
  RTC_FEATURE_FLAG_CURVES = ... ,

  RTC_FEATURE_FLAG_CONE_LINEAR_CURVE = 1 << 5,
  RTC_FEATURE_FLAG_ROUND_LINEAR_CURVE  = 1 << 6,
  RTC_FEATURE_FLAG_FLAT_LINEAR_CURVE = 1 << 7,

  RTC_FEATURE_FLAG_ROUND_BEZIER_CURVE = 1 << 8,
  RTC_FEATURE_FLAG_FLAT_BEZIER_CURVE = 1 << 9,
  RTC_FEATURE_FLAG_NORMAL_ORIENTED_BEZIER_CURVE = 1 << 10,

  RTC_FEATURE_FLAG_ROUND_BSPLINE_CURVE = 1 << 11,
  RTC_FEATURE_FLAG_FLAT_BSPLINE_CURVE = 1 << 12,
  RTC_FEATURE_FLAG_NORMAL_ORIENTED_BSPLINE_CURVE = 1 << 13,

  RTC_FEATURE_FLAG_ROUND_HERMITE_CURVE = 1 << 14,
  RTC_FEATURE_FLAG_FLAT_HERMITE_CURVE = 1 << 15,
  RTC_FEATURE_FLAG_NORMAL_ORIENTED_HERMITE_CURVE = 1 << 16,

  RTC_FEATURE_FLAG_ROUND_CATMULL_ROM_CURVE = 1 << 17,
  RTC_FEATURE_FLAG_FLAT_CATMULL_ROM_CURVE = 1 << 18,
  RTC_FEATURE_FLAG_NORMAL_ORIENTED_CATMULL_ROM_CURVE = 1 << 19,

  RTC_FEATURE_FLAG_SPHERE_POINT = 1 << 20,
  RTC_FEATURE_FLAG_DISC_POINT = 1 << 21,
  RTC_FEATURE_FLAG_ORIENTED_DISC_POINT = 1 << 22,

  RTC_FEATURE_FLAG_ROUND_CURVES = ... ,
  RTC_FEATURE_FLAG_FLAT_CURVES = ... ,
  RTC_FEATURE_FLAG_NORMAL_ORIENTED_CURVES = ... ,

  RTC_FEATURE_FLAG_LINEAR_CURVES = ... ,
  RTC_FEATURE_FLAG_BEZIER_CURVES = ... ,
  RTC_FEATURE_FLAG_BSPLINE_CURVES = ... ,
  RTC_FEATURE_FLAG_HERMITE_CURVES = ... ,
```

```
  RTC_FEATURE_FLAG_INSTANCE = 1 << 23,

  RTC_FEATURE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS = 1 << 24,
  RTC_FEATURE_FLAG_FILTER_FUNCTION_IN_GEOMETRY = 1 << 25,
  RTC_FEATURE_FLAG_FILTER_FUNCTION = ... ,

  RTC_FEATURE_FLAG_USER_GEOMETRY_CALLBACK_IN_ARGUMENTS = 1 << 26,
  RTC_FEATURE_FLAG_USER_GEOMETRY_CALLBACK_IN_GEOMETRY = 1 << 27,
  RTC_FEATURE_FLAG_USER_GEOMETRY = ... ,

  RTC_FEATURE_FLAG_32_BIT_RAY_MASK = 1 << 28,

  RTC_FEATURE_FLAG_ALL = 0xffffffff
};
```

DESCRIPTION

The `RTCFeatureFlags` enum specify a bit mask to enable specific ray tracing features for ray query operations. The feature flags are passed to the `rtcIntersect1/4/8/16` and `rtcOccluded1/4/8/16` functions through the `RTCIntersectArguments` and `RTCOccludedArguments` structures. Only a ray tracing feature whose bit is enabled in the feature mask can get used. If a feature bit is not set, the behaviour is undefined, thus the feature may work or not. To enable multiple features the respective features have to get combined using a bitwise `OR` operation.

The purpose of feature flags is to reduce code size on the GPU by enabling just the features required to render the scene. On the CPU there is no need to use feature flags, and the default of all features enabled (`RTC_FEATURE_FLAG_ALL`) can just be kept.

The following features can get enabled using feature flags:

- RTC_FEATURE_FLAG_MOTION_BLUR: Enables motion blur for all geometry types.
- RTC_FEATURE_FLAG_TRIANGLE: Enables triangle geometries (RTC_GEOMETRY_TYPE_TRIANGLE).
- RTC_FEATURE_FLAG_QUAD: Enables quad geometries (RTC_GEOMETRY_TYPE_QUAD).
- RTC_FEATURE_FLAG_GRID: Enables grid geometries (RTC_GEOMETRY_TYPE_GRID).
- RTC_FEATURE_FLAG_SUBDIVISION: Enables subdivision geometries (RTC_GEOMETRY_TYPE_SUBDIVISION).
- RTC_FEATURE_FLAG_POINT: Enables all point geometry types (RTC_GEOMETRY_TYPE_XXX_POINT)
- RTC_FEATURE_FLAG_CURVES: Enables all curve geometry types (RTC_GEOMETRY_TYPE_XXX_YYY_CURVE)
- RTC_FEATURE_FLAG_ROUND_CURVES: Enables all round curves (RTC_GEOMETRY_TYPE_ROUND_XXX_CURVE).
- RTC_FEATURE_FLAG_FLAT_CURVES: Enables all flat curves (RTC_GEOMETRY_TYPE_FLAT_XXX_CURVE).
- RTC_FEATURE_FLAG_NORMAL_ORIENTED_CURVES: Enables all normal oriented curves (RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_XXX_CURVE).
- RTC_FEATURE_FLAG_LINEAR_CURVES: Enables all linear curves (RTC_GEOMETRY_TYPE_XXX_LINEAR_CURVE).
- RTC_FEATURE_FLAG_BEZIER_CURVES: Enables all Bézier curves (RTC_GEOMETRY_TYPE_XXX_BEZIER_CURVE).
- RTC_FEATURE_FLAG_BSPLINE_CURVES: Enables all B-spline curves (RTC_GEOMETRY_TYPE_XXX_BSPLINE_CURVE).

- RTC_FEATURE_FLAG_HERMITE_CURVES: Enables all Hermite curves (RTC_GEOMETRY_TYPE_XXX_HERMITE_CURVE).
- RTC_FEATURE_FLAG_CONE_LINEAR_CURVE: Enables cone geometry type (RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE).
- RTC_FEATURE_FLAG_ROUND_LINEAR_CURVE: Enables round linear curves (RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE).
- RTC_FEATURE_FLAG_FLAT_LINEAR_CURVE: Enables flat linear curves (RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE).
- RTC_FEATURE_FLAG_ROUND_BEZIER_CURVE: Enables round Bézier curves (RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE).
- RTC_FEATURE_FLAG_FLAT_BEZIER_CURVE: Enables flat Bézier curves (RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE).
- RTC_FEATURE_FLAG_NORMAL_ORIENTED_BEZIER_CURVE: Enables normal oriented Bézier curves (RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE).
- RTC_FEATURE_FLAG_ROUND_BSPLINE_CURVE: Enables round B-spline curves (RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE).
- RTC_FEATURE_FLAG_FLAT_BSPLINE_CURVE: Enables flat B-spline curves (RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE).
- RTC_FEATURE_FLAG_NORMAL_ORIENTED_BSPLINE_CURVE: Enables normal oriented B-spline curves (RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE).
- RTC_FEATURE_FLAG_ROUND_HERMITE_CURVE: Enables round Hermite curves (RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE).
- RTC_FEATURE_FLAG_FLAT_HERMITE_CURVE: Enables flat Hermite curves (RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE).
- RTC_FEATURE_FLAG_NORMAL_ORIENTED_HERMITE_CURVE: Enables normal oriented Hermite curves (RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE).
- RTC_FEATURE_FLAG_ROUND_CATMULL_ROM_CURVE: Enables round Catmull Rom curves (RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE).
- RTC_FEATURE_FLAG_FLAT_CATMULL_ROM_CURVE: Enables flat Catmull Rom curves (RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE).
- RTC_FEATURE_FLAG_NORMAL_ORIENTED_CATMULL_ROM_CURVE: Enables normal oriented Catmull Rom curves (RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE).
- RTC_FEATURE_FLAG_SPHERE_POINT: Enables sphere geometry type (RTC_GEOMETRY_TYPE_SPHERE_POINT).
- RTC_FEATURE_FLAG_DISC_POINT: Enables disc geometry type (RTC_GEOMETRY_TYPE_DISC_POINT).
- RTC_FEATURE_FLAG_ORIENTED_DISC_POINT: Enables oriented disc geometry types (RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT).
- RTC_FEATURE_FLAG_INSTANCE: Enables instance geometries (RTC_GEOMETRY_TYPE_INSTANCE).
- RTC_FEATURE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS: Enables filter functions passed through intersect arguments.
- RTC_FEATURE_FLAG_FILTER_FUNCTION_IN_GEOMETRY: Enable filter functions passed through geometry.

- RTC_FEATURE_FLAG_FILTER_FUNCTION: Enables filter functions (argument and geometry version).
- RTC_FEATURE_FLAG_USER_GEOMETRY_CALLBACK_IN_ARGUMENTS: Enables RTC_GEOMETRY_TYPE_USER with function pointer passed through intersect arguments.
- RTC_FEATURE_FLAG_USER_GEOMETRY_CALLBACK_IN_GEOMETRY: Enables RTC_GEOMETRY_TYPE_USER with function pointer passed through geometry object.
- RTC_FEATURE_FLAG_USER_GEOMETRY: Enables RTC_GEOMETRY_TYPE_USER geometries (both argument and geometry callback versions).
- RTC_FEATURE_FLAG_32_BIT_RAY_MASK: Enables full 32 bit ray masks. If not used, only the lower 7 bits in the ray mask are handled correctly.
- RTC_FEATURE_FLAG_ALL: Enables all features (default).

EXIT STATUS

SEE ALSO

rtcIntersect1, rtcIntersect4/8/16, rtcOccluded1, rtcOccluded4/8/16,

7.99 rtcInitIntersectArguments

NAME

`rtcInitIntersectArguments - initializes the intersect arguments struct`

SYNOPSIS

```
#include <embree4/rtcore.h>

enum RTCRayQueryFlags
{
  RTC_RAY_QUERY_FLAG_NONE,
  RTC_RAY_QUERY_FLAG_INCOHERENT,
  RTC_RAY_QUERY_FLAG_COHERENT,
  RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER
};

struct RTCIntersectArguments
{
  enum RTCRayQueryFlags flags;
  enum RTCFeatureFlags feature_mask;
  struct RTCRayQueryContext* context;
  RTCFilterFunctionN filter;
  RTCIntersectFunctionN intersect;
#if RTC_MIN_WIDTH
  float minWidthDistanceFactor;
#endif
};

void rtcInitIntersectArguments(
  struct RTCIntersectArguments* args
);
```

DESCRIPTION

The `rtcInitIntersectArguments` function initializes the optional argument struct that can get passed to the `rtcIntersect1/4/8/16` functions to default values. The arguments struct needs to get used for more advanced Embree features as described here.

The `flags` member can get used to enable special traversal mode. Using the `RTC_RAY_QUERY_FLAG_INCOHERENT` flag uses an optimized traversal algorithm for incoherent rays (default), while `RTC_RAY_QUERY_FLAG_COHERENT` uses an optimized traversal algorithm for coherent rays (e.g. primary camera rays).

The `feature_mask` member should get used in SYCL to just enable ray tracing features required to render a given scene. Please see section RTCFeatureFlags for a more detailed description.

The `context` member can get used to pass an optional intersection context. It is guaranteed that the pointer to the context passed to a ray query is directly passed to all callback functions. This way it is possible to attach arbitrary data to the end of the context, such as a per-ray payload. Please note that the ray pointer is not guaranteed to be passed to the callback functions, thus reading additional data from the ray pointer passed to callbacks is not possible. See section rtcInitRayQueryContext for more details.

The `filter` member specifies a filter function to invoke for each encountered hit. The support for the argument filter function must be enabled for a scene by

using the RTC_SCENE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS scene flag. In case of instancing this feature has to get enabled also for each instantiated scene.

The argument filter function is invoked for each geometry for which it got explicitely enabled using the rtcSetGeometryEnableFilterFunctionFromArguments function. The invokation of the argument filter function can also get enfored for each geometry using the RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER ray query flag. This argument filter function is invoked as a second filter stage after the per-geometry filter function is invoked. Only rays that passed the first filter stage are valid in this second filter stage. Having such a per ray-query filter function can be useful to implement modifications of the behavior of the query, such as collecting all hits or accumulating transparencies.

The intersect member specifies the user geometry callback to get invoked for each user geometry encountered during traversal. The user geometry callback specified this way has preference over the one specified inside the geometry.

The minWidthDistanceFactor value controls the target size of the curve radii when the min-width feature is enabled. Please see the rtcSetGeometryMaxRadiusScale function for more details on the min-width feature.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcIntersect1, rtcIntersect4/8/16, RTCFeatureFlags, rtcInitRayQueryContext, RTC_GEOMETRY_TYPE_USER, rtcSetGeometryMaxRadiusScale

7.100 rtcInitOccludedArguments

NAME

rtcInitOccludedArguments - initializes the occluded arguments struct

SYNOPSIS

```
#include <embree4/rtcore.h>

enum RTCRayQueryFlags
{
  RTC_RAY_QUERY_FLAG_NONE,
  RTC_RAY_QUERY_FLAG_INCOHERENT,
  RTC_RAY_QUERY_FLAG_COHERENT,
  RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER
};

struct RTCOccludedArguments
{
  enum RTCRayQueryFlags flags;
  enum RTCFeatureFlags feature_mask;
  struct RTCRayQueryContext* context;
  RTCFilterFunctionN filter;
  RTCOccludedFunctionN intersect;
#if RTC_MIN_WIDTH
  float minWidthDistanceFactor;
#endif
};

void rtcInitOccludedArguments(
  struct RTCOccludedArguments* args
);
```

DESCRIPTION

The `rtcInitOccludedArguments` function initializes the optional argument struct that can get passed to the `rtcOccluded1/4/8/16` functions to default values. The arguments struct needs to get used for more advanced Embree features as described here.

The `flags` member can get used to enable special traversal mode. Using the `RTC_RAY_QUERY_FLAG_INCOHERENT` flag uses an optimized traversal algorithm for incoherent rays (default), while `RTC_RAY_QUERY_FLAG_COHERENT` uses an optimized traversal algorithm for coherent rays (e.g. primary camera rays).

The `feature_mask` member should get used in SYCL to just enable ray tracing features required to render a given scene. Please see section RTCFeatureFlags for a more detailed description.

The `context` member can get used to pass an optional intersection context. It is guaranteed that the pointer to the context passed to a ray query is directly passed to all callback functions. This way it is possible to attach arbitrary data to the end of the context, such as a per-ray payload. Please note that the ray pointer is not guaranteed to be passed to the callback functions, thus reading additional data from the ray pointer passed to callbacks is not possible. See section rtcInitRayQueryContext for more details.

The `filter` member specifies a filter function to invoked for each encountered hit. The support for the argument filter function must be enabled for a scene by using the `RTC_SCENE_FLAG_FILTER_FUNCTION_IN_ARGUMENTS` scene

flag. In case of instancing this feature has to get enabled also for each instantiated scene.

The argument filter function is invoked for each geometry for which it got explicitely enabled using the `rtcSetGeometryEnableFilterFunctionFromArguments` function. The invokation of the argument filter function can also get enfored for each geometry using the `RTC_RAY_QUERY_FLAG_INVOKE_ARGUMENT_FILTER` ray query flag. This argument filter function is invoked as a second filter stage after the per-geometry filter function is invoked. Only rays that passed the first filter stage are valid in this second filter stage. Having such a per ray-query filter function can be useful to implement modifications of the behavior of the query, such as collecting all hits or accumulating transparencies.

The `intersect` member specifies the user geometry callback to get invoked for each user geometry encountered during traversal. The user geometry callback specified this way has preference over the one specified inside the geometry.

The `minWidthDistanceFactor` value controls the target size of the curve radii when the min-width feature is enabled. Please see the rtcSetGeometryMaxRadiusScale function for more details on the min-width feature.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcOccluded1, rtcOccluded4/8/16, RTCFeatureFlags, rtcInitRayQueryContext, RTC_GEOMETRY_TYPE_USER, rtcSetGeometryMaxRadiusScale

7.101 rtcInitRayQueryContext

NAME

`rtcInitRayQueryContext - initializes the ray query context`

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCRayQueryContext
{
  #if RTC_MAX_INSTANCE_LEVEL_COUNT > 1
    unsigned int instStackSize;
  #endif

  unsigned int instID[RTC_MAX_INSTANCE_LEVEL_COUNT];
};

void rtcInitRayQueryContext(
  struct RTCRayQueryContext* context
);
```

DESCRIPTION

The `rtcInitRayQueryContext` function initializes the intersection context to default values and should be called to initialize every ray query context.

It is guaranteed that the pointer to the ray query context (`RTCRayQueryContext` type) is passed to the registered callback functions. This way it is possible to attach arbitrary data to the end of the ray query context, such as a per-ray payload.

Inside the user geometry callback the ray query context can get used to access the `instID` stack to know which instance the user geometry object resides.

If not ray query context is specified when tracing a ray, a default context is used.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcIntersect1, rtcIntersect4/8/16, rtcOccluded1, rtcOccluded4/8/16

7.102 rtcIntersect1

NAME

```
rtcIntersect1 - finds the closest hit for a single ray
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcIntersect1(
  RTCScene scene,
  struct RTCRayHit* rayhit
  struct RTCIntersectArguments* args = NULL
);
```

DESCRIPTION

The `rtcIntersect1` function finds the closest hit of a single ray (`rayhit` argument) with the scene (`scene` argument). The provided ray/hit structure contains the ray to intersect and some hit output fields that are filled when a hit is found. The passed optional arguments struct (`args` argument) can get used for advanced use cases, see section rtcInitIntersectArguments for more details.

To trace a ray, the user has to initialize the ray origin (`org` ray member), ray direction (`dir` ray member), ray segment (`tnear`, `tfar` ray members), ray mask (`mask` ray member), and set the ray flags to 0 (`flags` ray member). The ray time (`time` ray member) must be initialized to a value in the range $[0, 1]$. 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. See Section RTCRay for the ray layout description.

The geometry ID (`geomID` hit member) of the hit data must be initialized to `RTC_INVALID_GEOMETRY_ID` (-1).

When no intersection is found, the ray/hit data is not updated. When an intersection is found, the hit distance is written into the `tfar` member of the ray and all hit data is set, such as unnormalized geometry normal in object space (`Ng` hit member), local hit coordinates (`u`, `v` hit member), instance ID stack (`instID` hit member), geometry ID (`geomID` hit member), and primitive ID (`primID` hit member). See Section RTCHit for the hit layout description.

If the instance ID stack has a prefix of values not equal to `RTC_INVALID_GEOMETRY_ID`, the instance ID on each level corresponds to the geometry ID of the hit instance of the higher-level scene, the geometry ID corresponds to the hit geometry inside the hit instanced scene, and the primitive ID corresponds to the n-th primitive of that geometry.

If level 0 of the instance ID stack is equal to `RTC_INVALID_GEOMETRY_ID`, the geometry ID corresponds to the hit geometry inside the top-level scene, and the primitive ID corresponds to the n-th primitive of that geometry.

The implementation makes no guarantees that 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 slightly enlarged `tnear`, and if you want to include intersections at `tfar` pass a slightly enlarged `tfar`.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the ray query context. See section rtcInitRayQueryContext for more details.

The ray/hit structure must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded1, rtcIntersect4/8/16, RTCRayHit, rtcInitIntersectArguments

7.103 rtcOccluded1

NAME

`rtcOccluded1 - finds any hit for a single ray`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcOccluded1(
  RTCScene scene,
  struct RTCRay* ray,
  struct RTCOccludedArguments* args = NULL
);
```

DESCRIPTION

The `rtcOccluded1` function checks for a single ray (`ray` argument) whether there is any hit with the scene (`scene` argument). The passed optional arguments struct (`args` argument) can get used for advanced use cases, see section rtcInitOccludedArguments for more details.

To trace a ray, the user must initialize the ray origin (`org` ray member), ray direction (`dir` ray member), ray segment (`tnear`, `tfar` ray members), ray mask (`mask` ray member), and must set the ray flags to `0` (`flags` ray member). The ray time (`time` ray member) must be initialized to a value in the range $[0, 1]$. The ray segment must be in the range $[0, \infty]$, thus ranges that start behind the ray origin are not valid, but ranges can reach to infinity. See Section RTCRay for the ray layout description.

When no intersection is found, the ray data is not updated. In case a hit was found, the `tfar` component of the ray is set to `-inf`.

The implementation makes no guarantees that 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 slightly enlarged `tnear`, and if you want to include intersections at `tfar` pass a slightly enlarged `tfar`.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the ray query context. See section rtcInitRayQueryContext for more details.

The ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersect1, rtcOccluded4/8/16, RTCRay, rtcInitOccludedArguments

7.104 rtcIntersect4/8/16

NAME

`rtcIntersect4/8/16 - finds the closest hits for a ray packet`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcIntersect4(
  const int* valid,
  RTCScene scene,
  struct RTCRayHit4* rayhit,
  struct RTCIntersectArguments* args = NULL
);

void rtcIntersect8(
  const int* valid,
  RTCScene scene,
  struct RTCRayHit8* rayhit,
  struct RTCIntersectArguments* args = NULL
);

void rtcIntersect16(
  const int* valid,
  RTCScene scene,
  struct RTCRayHit16* rayhit,
  struct RTCIntersectArguments* args = NULL
);
```

DESCRIPTION

The `rtcIntersect4/8/16` functions finds the closest hits for a ray packet of size 4, 8, or 16 (`rayhit` argument) with the scene (`scene` argument). The ray/hit input contains a ray packet and hit packet. The passed optional arguments struct (`args` argument) are used to pass additional arguments for advanced features. See Section rtcIntersect1 for more details and a description of how to set up and trace rays.

A ray valid mask must be provided (`valid` argument) which stores one 32-bit integer (`-1` means valid and `0` invalid) per ray in the packet. Only active rays are processed, and hit data of inactive rays is not changed.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the ray query context. See section rtcInitRayQueryContext for more details.

For `rtcIntersect4` the ray packet must be aligned to 16 bytes, for `rtcIntersect8` the alignment must be 32 bytes, and for `rtcIntersect16` the alignment must be 64 bytes.

The `rtcIntersect4`, `rtcIntersect8` and `rtcIntersect16` functions may change the ray packet size and ray order when calling back into filter functions or user geometry callbacks. Under some conditions the application can assume packets to stay intakt, which can determined by querying the `RTC_DEVICE_PROPERTY_NATIVE_RAY4_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY8_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY16_SUPPORTED` properties through the `rtcGetDeviceProperty` function. See rtcGetDeviceProperty for more information.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersect1, rtcOccluded4/8/16, rtcInitIntersectArguments

7.105 rtcOccluded4/8/16

NAME

`rtcOccluded4/8/16 - finds any hits for a ray packet`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcOccluded4(
  const int* valid,
  RTCScene scene,
  struct RTCRay4* ray,
  struct RTCOccludedArguments* args = NULL
);

void rtcOccluded8(
  const int* valid,
  RTCScene scene,
  struct RTCRay8* ray,
  struct RTCOccludedArguments* args = NULL
);

void rtcOccluded16(
  const int* valid,
  RTCScene scene,
  struct RTCRay16* ray,
  struct RTCOccludedArguments* args = NULL
);
```

DESCRIPTION

The `rtcOccluded4/8/16` functions checks for each active ray of the ray packet of size 4, 8, or 16 (`ray` argument) whether there is any hit with the scene (`scene` argument). The passed optional arguments struct (`args` argument) can get used for advanced use cases, see section rtcInitOccludedArguments for more details. See Section rtcOccluded1 for more details and a description of how to set up and trace occlusion rays.

A ray valid mask must be provided (`valid` argument) which stores one 32-bit integer (`-1` means valid and `0` invalid) per ray in the packet. Only active rays are processed, and hit data of inactive rays is not changed.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the ray query context. See section rtcInitRayQueryContext for more details.

For `rtcOccluded4` the ray packet must be aligned to 16 bytes, for `rtcOccluded8` the alignment must be 32 bytes, and for `rtcOccluded16` the alignment must be 64 bytes.

The `rtcOccluded4`, `rtcOccluded8` and `rtcOccluded16` functions may change the ray packet size and ray order when calling back into intersect filter functions or user geometry callbacks. Under some conditions the application can assume packets to stay intakt, which can determined by querying the `RTC_DEVICE_PROPERTY_NATIVE_RAY4_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY8_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY16_SUPPORTED` properties through the `rtcGetDeviceProperty` function. See rtcGetDeviceProperty for more information.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded1, rtcIntersect4/8/16, rtcInitOccludedArguments

7.106 rtcForwardIntersect1

NAME

```
rtcForwardIntersect1 - forwards a single ray to new scene
  from user geometry callback
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcForwardIntersect1(
  const struct RTCIntersectFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay* ray,
  unsigned int instID
);
```

DESCRIPTION

The `rtcForwardIntersect1` function forwards the traversal of a transformed ray (`ray` argument) into a scene (`scene` argument) from a user geometry callback. The function can only get invoked from a user geometry callback for a ray traversal initiated with the `rtcIntersect1` function. The callback arguments structure of the callback invokation has to get passed to the ray forwarding (`args` argument). The user geometry callback should instantly terminate after invoking the `rtcForwardIntersect1` function.

Only the ray origin and ray direction members of the ray argument are used for forwarding, all additional ray properties are inherited from the initial ray traversal invokation of `rtcIntersect1`.

The implementation of the `rtcForwardIntersect1` function recursively continues the ray traversal into the specified scene and pushes the provided instance ID (`instID` argument) to the instance ID stack. Hit information is updated into the ray hit structure passed to the original `rtcIntersect1` invokation.

This function can get used to implement user defined instancing using user geometries, e.g. by transforming the ray in a special way, and/or selecting between different scenes to instantiate.

When using Embree on the CPU it is possible to recursively invoke `rtcIntersect1` directly from a user geometry callback. However, when SYCL is used, recursively tracing rays is not directly supported, and the `rtcForwardIntersect1` function must be used.

The ray structure must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersect1, RTCRay

7.107 rtcForwardOccluded1

NAME

```
rtcForwardOccluded1 - forwards a single ray to new scene
  from user geometry callback
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcForwardOccluded1(
  const struct RTCOccludedFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay* ray,
  unsigned int instID
);
```

DESCRIPTION

The `rtcForwardOccluded1` function forwards the traversal of a transformed ray (`ray` argument) into a scene (`scene` argument) from a user geometry callback. The function can only get invoked from a user geometry callback for a ray traversal initiated with the `rtcOccluded1` function. The callback arguments structure of the callback invokation has to get passed to the ray forwarding (`args` argument). The user geometry callback should instantly terminate after invoking the `rtcForwardOccluded1` function.

Only the ray origin and ray direction members of the ray argument are used for forwarding, all additional ray properties are inherited from the initial ray traversal invokation of `rtcOccluded1`.

The implementation of the `rtcForwardOccluded1` function recursively continues the ray traversal into the specified scene and pushes the provided instance ID (`instID` argument) to the instance ID stack. Hit information is updated into the ray structure passed to the original `rtcOccluded1` invokation.

This function can get used to implement user defined instancing using user geometries, e.g. by transforming the ray in a special way, and/or selecting between different scenes to instantiate.

When using Embree on the CPU it is possible to recursively invoke `rtcOccluded1` directly from a user geometry callback. However, when SYCL is used, recursively tracing rays is not directly supported, and the `rtcForwardOccluded1` function must be used.

The ray structure must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded1, RTCRay

7.108 rtcForwardIntersect4/8/16

NAME

```
rtcForwardIntersect4/8/16 - forwards a ray packet to new scene
  from user geometry callback
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcForwardIntersect4(
  void int* valid,
  const struct RTCIntersectFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay4* ray,
  unsigned int instID
);

void rtcForwardIntersect4(
  void int* valid,
  const struct RTCIntersectFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay4* ray,
  unsigned int instID
);

void rtcForwardIntersect16(
  void int* valid,
  const struct RTCIntersectFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay16* ray,
  unsigned int instID
);
```

DESCRIPTION

The `rtcForwardIntersect4/8/16` functions forward the traversal of a transformed ray packet (`ray` argument) into a scene (`scene` argument) from a user geometry callback. The function can only get invoked from a user geometry callback for a ray traversal initiated with the `rtcIntersect4/8/16` function. The callback arguments structure of the callback invokation has to get passed to the ray forwarding (`args` argument). The user geometry callback should instantly terminate after invoking the `rtcForwardIntersect4/8/16` function.

Only the ray origin and ray direction members of the ray argument are used for forwarding, all additional ray properties are inherited from the initial ray traversal invokation of `rtcIntersect4/8/16`.

The implementation of the `rtcForwardIntersect4/8/16` function recursively continues the ray traversal into the specified scene and pushes the provided instance ID (`instID` argument) to the instance ID stack. Hit information is updated into the ray hit structure passed to the original `rtcIntersect4/8/16` invokation.

This function can get used to implement user defined instancing using user geometries, e.g. by transforming the ray in a special way, and/or selecting between different scenes to instantiate.

When using Embree on the CPU it is possible to recursively invoke `rtcIntersect4/8/16` directly from a user geometry callback. However, when SYCL

is used, recursively tracing rays is not directly supported, and the `rtcForwardIntersect4/8/16` function must be used.

For `rtcForwardIntersect4` the ray packet must be aligned to 16 bytes, for `rtcForwardIntersect8` the alignment must be 32 bytes, and for `rtcForwardIntersect16` the alignment must be 64 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersect4/8/16

7.109 rtcForwardOccluded4/8/16

NAME

```
rtcForwardOccluded4/8/16 - forwards a ray packet to new scene
  from user geometry callback
```

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcForwardOccluded4(
  void int* valid,
  const struct RTCOccludedFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay4* ray,
  unsigned int instID
);

void rtcForwardOccluded4(
  void int* valid,
  const struct RTCOccludedFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay4* ray,
  unsigned int instID
);

void rtcForwardOccluded16(
  void int* valid,
  const struct RTCOccludedFunctionNArguments* args,
  RTCScene scene,
  struct RTCRay16* ray,
  unsigned int instID
);
```

DESCRIPTION

The `rtcForwardOccluded4/8/16` functions forward the traversal of a transformed ray packet (`ray` argument) into a scene (`scene` argument) from a user geometry callback. The function can only get invoked from a user geometry callback for a ray traversal initiated with the `rtcOccluded4/8/16` function. The callback arguments structure of the callback invokation has to get passed to the ray forwarding (`args` argument). The user geometry callback should instantly terminate after invoking the `rtcForwardOccluded4/8/16` function.

Only the ray origin and ray direction members of the ray argument are used for forwarding, all additional ray properties are inherited from the initial ray traversal invokation of `rtcOccluded4/8/16`.

The implementation of the `rtcForwardOccluded4/8/16` function recursively continues the ray traversal into the specified scene and pushes the provided instance ID (`instID` argument) to the instance ID stack. Hit information is updated into the ray structure passed to the original `rtcOccluded4/8/16` invokation.

This function can get used to implement user defined instancing using user geometries, e.g. by transforming the ray in a special way, and/or selecting between different scenes to instantiate.

When using Embree on the CPU it is possible to recursively invoke `rtcOccluded4/8/16` directly from a user geometry callback. However, when SYCL is

used, recursively tracing rays is not directly supported, and the `rtcForwardOccluded4/8/16` function must be used.

For `rtcForwardOccluded4` the ray packet must be aligned to 16 bytes, for `rtcForwardOccluded8` the alignment must be 32 bytes, and for `rtcForwardOccluded16` the alignment must be 64 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded4/8/16

7.110 rtcInitPointQueryContext

NAME

```
rtcInitPointQueryContext - initializes the context information (e.g.
  stack of (multilevel-)instance transformations) for point queries
```

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTC_ALIGN(16) RTCPointQueryContext
{
  // accumulated 4x4 column major matrices from world to instance space.
  float world2inst[RTC_MAX_INSTANCE_LEVEL_COUNT][16];

  // accumulated 4x4 column major matrices from instance to world space.
  float inst2world[RTC_MAX_INSTANCE_LEVEL_COUNT][16];

  // instance ids.
  unsigned int instID[RTC_MAX_INSTANCE_LEVEL_COUNT];

  // number of instances currently on the stack.
  unsigned int instStackSize;
};

void rtcInitPointQueryContext(
  struct RTCPointQueryContext* context
);
```

DESCRIPTION

A stack (`RTCPointQueryContext` type) which stores the IDs and instance transformations during a BVH traversal for a point query. The transformations are assumed to be affine transformations (3×3 matrix plus translation) and therefore the last column is ignored (see RTC_GEOMETRY_TYPE_INSTANCE for details).

The `rtcInitPointContext` function initializes the context to default values and should be called for initialization.

The context will be passed as an argument to the point query callback function (see rtcSetGeometryPointQueryFunction) and should be used to pass instance information down the instancing chain for user defined instancing (see tutorial [ClosestPoint] for a reference implementation of point queries with user defined instancing).

The context is an necessary argument to rtcPointQuery and Embree internally uses the topmost instance transformation of the stack to transform the point query into instance space.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcPointQuery, rtcSetGeometryPointQueryFunction

7.111 rtcPointQuery

NAME

`rtcPointQuery - traverses the BVH with a point query object`

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTC_ALIGN(16) RTCPointQuery
{
  // location of the query
  float x;
  float y;
  float z;

  // radius and time of the query
  float radius;
  float time;
};

void rtcPointQuery(
  RTCScene scene,
  struct RTCPointQuery* query,
  struct RTCPointQueryContext* context,
  struct RTCPointQueryFunction* queryFunc,
  void* userPtr
);
```

DESCRIPTION

The `rtcPointQuery` function traverses the BVH using a `RTCPointQuery` object (`query` argument) and calls a user defined callback function (e.g `queryFunc` argument) for each primitive of the scene (`scene` argument) that intersects the query domain.

The user has to initialize the query location (`x`, `y` and `z` member) and query radius in the range $[0, \infty]$. If the scene contains motion blur geometries, also the query time (`time` member) must be initialized to a value in the range $[0, 1]$.

Further, a `RTCPointQueryContext` (`context` argument) must be created and initialized. It contains ID and transformation information of the instancing hierarchy if (multilevel-)instancing is used. See rtcInitPointQueryContext for further information.

For every primitive that intersects the query domain, the callback function (`queryFunc` argument) is called, in which distance computations to the primitive can be implemented. The user will be provided with the primID and geomID of the according primitive, however, the geometry information (e.g. triangle index and vertex data) has to be determined manually. The `userPtr` argument can be used to input geometry data of the scene or output results of the point query (e.g. closest point currently found on surface geometry (see tutorial [ClosestPoint])).

The parameter `queryFunc` is optional and can be NULL, in which case the callback function is not invoked. However, a callback function can still get attached to a specific `RTCGeometry` object using rtcSetGeometryPointQueryFunction. If a callback function is attached to a geometry and (a potentially different) callback function is passed as an argument to `rtcPointQuery`, both functions are called for the primitives of the according geometries.

The query radius can be decreased inside the callback function, which allows to efficiently cull parts of the scene during BVH traversal. Increasing the query radius and modifying time or location of the query will result in undefined behaviour.

The callback function will be called for all primitives in a leaf node of the BVH even if the primitive is outside the query domain, since Embree does not gather geometry information of primitives internally.

Point queries can be used with (multilevel)-instancing. However, care has to be taken when the instance transformation contains anisotropic scaling or sheering. In these cases distance computations have to be performed in world space to ensure correctness and the ellipsoidal query domain (in instance space) will be approximated with its axis aligned bounding box internally. Therefore, the callback function might be invoked even for primitives in inner BVH nodes that do not intersect the query domain. See rtcSetGeometryPointQueryFunction for details.

The point query structure must be aligned to 16 bytes.

SUPPORTED PRIMITIVES

Currently, all primitive types are supported by the point query API except of points (see RTC_GEOMETRY_TYPE_POINT), curves (see RTC_GEOMETRY_TYPE_CURVE) and sudivision surfaces (see [RTC_GEOMETRY_SUBDIVISION]).

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcSetGeometryPointQueryFunction, rtcInitPointQueryContext

7.112 rtcCollide

NAME

`rtcCollide - intersects one BVH with another`

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTCCollision {
  unsigned int geomID0, primID0;
  unsigned int geomID1, primID1;
};

typedef void (*RTCCollideFunc) (
  void* userPtr,
  RTCCollision* collisions,
  size_t num_collisions);

void rtcCollide (
    RTCScene hscene0,
    RTCScene hscene1,
    RTCCollideFunc callback,
    void* userPtr
);
```

DESCRIPTION

The `rtcCollide` function intersects the BVH of `hscene0` with the BVH of scene `hscene1` and calls a user defined callback function (e.g `callback` argument) for each pair of intersecting primitives between the two scenes. A user defined data pointer (`userPtr` argument) can also be passed in.

For every pair of primitives that may intersect each other, the callback function (`callback` argument) is called. The user will be provided with the primID's and geomID's of multiple potentially intersecting primitive pairs. Currently, only scene entirely composed of user geometries are supported, thus the user is expected to implement a primitive/primitive intersection to filter out false positives in the callback function. The `userPtr` argument can be used to input geometry data of the scene or output results of the intersection query.

SUPPORTED PRIMITIVES

Currently, the only supported type is the user geometry type (see RTC_GEOMETRY_TYPE_USER).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

7.113 rtcNewBVH

NAME

```
rtcNewBVH - creates a new BVH object
```

SYNOPSIS

```
#include <embree4/rtcore.h>

RTCBVH rtcNewBVH(RTCDevice device);
```

DESCRIPTION

This function creates a new BVH object and returns a handle to this BVH. The BVH object is reference counted with an initial reference count of 1. The handle can be released using the `rtcReleaseBVH` API call.

The BVH object can be used to build a BVH in a user-specified format over user-specified primitives. See the documentation of the `rtcBuildBVH` call for more details.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainBVH, rtcReleaseBVH, rtcBuildBVH

7.114 rtcRetainBVH

NAME

`rtcRetainBVH - increments the BVH reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcRetainBVH(RTCBVH bvh);
```

DESCRIPTION

BVH objects are reference counted. The `rtcRetainBVH` function increments the reference count of the passed BVH object (`bvh` argument). This function together with `rtcReleaseBVH` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBVH, rtcReleaseBVH

7.115 rtcReleaseBVH

NAME

`rtcReleaseBVH - decrements the BVH reference count`

SYNOPSIS

```
#include <embree4/rtcore.h>

void rtcReleaseBVH(RTCBVH bvh);
```

DESCRIPTION

BVH objects are reference counted. The `rtcReleaseBVH` function decrements the reference count of the passed BVH object (`bvh` argument). When the reference count falls to 0, the BVH gets destroyed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBVH, rtcRetainBVH

7.116 rtcBuildBVH

NAME

rtcBuildBVH - builds a BVH

SYNOPSIS

```
#include <embree4/rtcore.h>

struct RTC_ALIGN(32) RTCBuildPrimitive
{
  float lower_x, lower_y, lower_z;
  unsigned int geomID;
  float upper_x, upper_y, upper_z;
  unsigned int primID;
};

typedef void* (*RTCCreateNodeFunction) (
  RTCThreadLocalAllocator allocator,
  unsigned int childCount,
  void* userPtr
);

typedef void (*RTCSetNodeChildrenFunction) (
  void* nodePtr,
  void** children,
  unsigned int childCount,
  void* userPtr
);

typedef void (*RTCSetNodeBoundsFunction) (
  void* nodePtr,
  const struct RTCBounds** bounds,
  unsigned int childCount,
  void* userPtr
);

typedef void* (*RTCCreateLeafFunction) (
  RTCThreadLocalAllocator allocator,
  const struct RTCBuildPrimitive* primitives,
  size_t primitiveCount,
  void* userPtr
);

typedef void (*RTCSplitPrimitiveFunction) (
  const struct RTCBuildPrimitive* primitive,
  unsigned int dimension,
  float position,
  struct RTCBounds* leftBounds,
  struct RTCBounds* rightBounds,
  void* userPtr
);

typedef bool (*RTCProgressMonitorFunction)(
  void* userPtr, double n
);
```

```
enum RTCBuildFlags
{
  RTC_BUILD_FLAG_NONE,
  RTC_BUILD_FLAG_DYNAMIC
};

struct RTCBuildArguments
{
  size_t byteSize;

  enum RTCBuildQuality buildQuality;
  enum RTCBuildFlags buildFlags;
  unsigned int maxBranchingFactor;
  unsigned int maxDepth;
  unsigned int sahBlockSize;
  unsigned int minLeafSize;
  unsigned int maxLeafSize;
  float traversalCost;
  float intersectionCost;

  RTCBVH bvh;
  struct RTCBuildPrimitive* primitives;
  size_t primitiveCount;
  size_t primitiveArrayCapacity;

  RTCCreateNodeFunction createNode;
  RTCSetNodeChildrenFunction setNodeChildren;
  RTCSetNodeBoundsFunction setNodeBounds;
  RTCCreateLeafFunction createLeaf;
  RTCSplitPrimitiveFunction splitPrimitive;
  RTCProgressMonitorFunction buildProgress;
  void* userPtr;
};

struct RTCBuildArguments rtcDefaultBuildArguments();

void* rtcBuildBVH(
  const struct RTCBuildArguments* args
);
```

DESCRIPTION

The `rtcBuildBVH` function can be used to build a BVH in a user-defined format over arbitrary primitives. All arguments to the function are provided through the `RTCBuildArguments` structure. The first member of that structure must be set to the size of the structure in bytes (`bytesSize` member) which allows future extensions of the structure. It is recommended to initialize the build arguments structure using the `rtcDefaultBuildArguments` function.

The `rtcBuildBVH` function gets passed the BVH to build (`bvh` member), the array of primitives (`primitives` member), the capacity of that array (`primitiveArrayCapacity` member), the number of primitives stored inside the array (`primitiveCount` member), callback function pointers, and a user-defined pointer (`userPtr` member) that is passed to all callback functions when invoked. The `primitives` array can be freed by the application after the BVH is built. All callback functions are typically called from multiple threads, thus their imple-

mentation must be thread-safe.

Four callback functions must be registered, which are invoked during build to create BVH nodes (`createNode` member), to set the pointers to all children (`setNodeChildren` member), to set the bounding boxes of all children (`setNodeBounds` member), and to create a leaf node (`createLeaf` member).

The function pointer to the primitive split function (`splitPrimitive` member) may be `NULL`, however, then no spatial splitting in high quality mode is possible. The function pointer used to report the build progress (`buildProgress` member) is optional and may also be `NULL`.

Further, some build settings are passed to configure the BVH build. Using the build quality settings (`buildQuality` member), 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 maximum branching factor of the BVH (`maxBranchingFactor` member), the maximum depth the BVH should have (`maxDepth` member), the block size for the SAH heuristic (`sahBlockSize` member), the minimum and maximum leaf size (`minLeafSize` and `maxLeafSize` member), and the estimated costs of one traversal step and one primitive intersection (`traversalCost` and `intersectionCost` members). When enabling the `RTC_BUILD_FLAG_DYNAMIC` build flags (`buildFlags` member), re-build performance for dynamic scenes is improved at the cost of higher memory requirements.

To spatially split primitives in high quality mode, the builder needs extra space at the end of the build primitive array to store split primitives. The total capacity of the build primitive array is passed using the `primitiveArrayCapacity` member, and should be about twice the number of primitives when using spatial splits.

The `RTCCreateNodeFunc` and `RTCCreateLeafFunc` callbacks are passed a thread local allocator object that should be used for fast allocation of nodes using the `rtcThreadLocalAlloc` function. We strongly recommend using this allocation mechanism, as alternative approaches like standard `malloc` can be over 10× slower. The allocator object passed to the create callbacks may be used only inside the current thread. Memory allocated using `rtcThreadLocalAlloc` is automatically freed when the `RTCBVH` object is deleted. If you use your own memory allocation scheme you have to free the memory yourself when the `RTCBVH` object is no longer used.

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

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

The `RTCSetNodeBoundsFunc` callback function gets 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 (`childCount` argument).

The `RTCCreateLeafFunc` callback additionally gets an array of primitives as input (`primitives` argument), and the size of this array (`primitiveCount` 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 (`primitive` argument) at the specified position (`position` argument) and dimension (`dimension` argument). The callback should return bounds of the clipped left and right parts of the primitive (`leftBounds` and `rightBounds` arguments).

The `RTCProgressMonitorFunction` callback function is called with the estimated completion rate `n` in the range $[0, 1]$. Returning `true` from the callback lets the build continue; returning `false` cancels the build.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBVH

7.117 RTCQuaternionDecomposition

NAME

```
RTCQuaternionDecomposition - structure that represents a quaternion
  decomposition of an affine transformation
```

SYNOPSIS

```
struct RTCQuaternionDecomposition
{
  float scale_x, scale_y, scale_z;
  float skew_xy, skew_xz, skew_yz;
  float shift_x, shift_y, shift_z;
  float quaternion_r, quaternion_i, quaternion_j, quaternion_k;
  float translation_x, translation_y, translation_z;
};
```

DESCRIPTION

The struct `RTCQuaternionDecomposition` represents an affine transformation decomposed into three parts. An upper triangular scaling/skew/shift matrix

$$S = \begin{pmatrix} scale_x & skew_{xy} & skew_{xz} & shift_x \\ 0 & scale_y & skew_{yz} & shift_y \\ 0 & 0 & scale_z & shift_z \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

a translation matrix

$$T = \begin{pmatrix} 1 & 0 & 0 & translation_x \\ 0 & 1 & 0 & translation_y \\ 0 & 0 & 1 & translation_z \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and a rotation matrix R, represented as a quaternion
$quaternion_r + quaternion_i\ \mathbf{i} + quaternion_j\ \mathbf{i} + quaternion_k\ \mathbf{k}$
where **i**, **j** **k** are the imaginary quaternion units. The passed quaternion will be normalized internally.

The affine transformation matrix corresponding to a `RTCQuaternionDecomposition` is TRS and a point $p = (p_x, p_y, p_z, 1)^T$ will be transformed as

$$p' = T\ R\ S\ p.$$

The functions `rtcInitQuaternionDecomposition`, `rtcQuaternionDecompositionSetQuaternion`, `rtcQuaternionDecompositionSetScale`, `rtcQuaternionDecompositionSetSkew`, `rtcQuaternionDecompositionSetShift`, and `rtcQuaternionDecompositionSetTranslation` allow to set the fields of the structure more conveniently.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcSetGeometryTransformQuaternion, rtcInitQuaternionDecomposition

7.118 rtcInitQuaternionDecomposition

NAME

`rtcInitQuaternionDecomposition - initializes quaternion decomposition`

SYNOPSIS

```
void rtcInitQuaternionDecomposition(
  struct RTCQuaternionDecomposition* qd
);
```

DESCRIPTION

The `rtcInitQuaternionDecomposition` function initializes a `RTCQuaternion-Decomposition` structure to represent an identity transformation.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcSetGeometryTransformQuaternion, RTCQuaternionDecomposition

Chapter 8

CPU Performance Recommendations

8.1 MXCSR control and status register

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`-type and `rtcOccluded`-type 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);
```

If using a different tasking system, make sure each rendering thread has the proper mode set.

8.2 Thread Creation and Affinity Settings

Tasking systems like TBB create worker threads on demand, which will add a runtime overhead for the very first `rtcCommitScene` call. In case you want to benchmark the scene build time, you should start the threads at application startup. You can let Embree start TBB threads by passing `start_threads=1` to the `cfg` 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 `cfg` parameter of `rtcNewDevice`. By default, threads are not affinitized by Embree with the exception of Xeon Phi Processors where they are affinitized by default.

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

8.3 Fast Coherent Rays

For getting the highest performance for highly coherent rays, e.g. primary or hard shadow rays, it is recommended to use packets with setting the `RTC_RAY_QUERY_FLAG_COHERENT` flag in the `RTCIntersectArguments` struct passed to the `rtcIntersect`/`rtcOccluded` calls. The rays inside each packet should be grouped as coherent as possible.

8.4 Huge Page Support

It is recommended to use huge pages under Linux to increase rendering performance. Embree supports 2MB huge pages under Windows, Linux, and macOS. Under Linux huge page support is enabled by default, and under Windows and macOS disabled by default. Huge page support can be 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 elevated mode which is a security issue, thus likely not an option for most use cases. Under macOS huge pages are rarely available as memory tends to get quickly fragmented, thus we do not recommend using huge pages on macOS.

8.4.1 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 4KB pages to 2MB pages when possible as a background job. Many Linux distributions have transparent huge pages enabled by default. See the following webpage for more information on transparent huge pages under Linux. In this mode each application, including your rendering application based on Embree, will automatically tend to use huge pages.

Using transparent huge pages, the transitioning from 4KB 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 be configured by writing some number of huge pages to allocate to `/proc/sys/vm/nr_overcommit_hugepages` as root user. E.g. to configure 2GB of address 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.

8.4.2 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 must be executed as an elevated process (“Run as administrator”) and the “SeLockMemoryPrivilege” must be explicitly 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 should be enabled in Embree by passing `hugepages=1` to `rtcNewDevice`.

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

8.4.3 Huge Pages under macOS

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

When the system has been running for a while, physical memory gets quickly fragmented, and causes huge page allocations to fail. For this reason, huge pages are not very useful under macOS in practice.

8.5 Avoid store-to-load forwarding issues with single rays

We recommend to use a single SSE store to set up the `org` and `tnear` components, and a single SSE store to set up the `dir` and `time` components of a single ray (`RTCRay` type). Storing these values using scalar stores causes a store-to-load forwarding penalty because Embree is reading these components using SSE loads later on.

Chapter 9

GPU Performance Recommendations

9.1 Low Code Complexity

As a general rule try to keep code complexity low, to avoid spill code generation. To achieve this we recommend splitting your renderer into separate kernels instead of using a single Uber kernel invokation.

Code can further get reduced by using SYCL specialization constants to just enable rendering features required to render a given scene.

9.2 Feature Flags

Use SYCL specialization constants and the feature flags (see section RTCFeatureFlags) of the `rtcIntersect1` and `rtcOccluded1` calls to JIT compile minimal code. The passed feature flags should just contain features required to render the current scene. If JIT compile times are an issue, reduce the number of feature masks used and use JIT caching (see section SYCL JIT caching).

9.3 Inline Indirect Calls

Attaching user geometry and intersection filter callbacks to the geometries of the scene is not supported in SYCL for performance reasons.

Instead directly pass the user geometry and intersection filter callback functions through the `RTCIntersectArguments` (and `RTCOccludedArguments`) struct to `rtcIntersect1` (and `rtcOccluded1`) API functions as in the following example:

```
RTC_SYCL_INDIRECTLY_CALLABLE void intersectionFilter(
  const RTCFilterFunctionNArguments* args
) { ... }

RTCIntersectArguments args;
rtcInitIntersectArguments(&args);
args.filter = intersectionFilter;

rtcIntersect1(scene,&ray,&args);
```

If the callback function is directly passed that way, the SYCL compiler can inline the indirect call, which gives a huge performance benefit. Do *not* read a

function pointer form some memory location and pass it to `rtcIntersect1` (and `rtcOccluded1`) as this will also prevent inlining.

9.4 7 Bit Ray Mask

Use just the lower 7 bits of the ray and geometry mask if possible, even though Embree supports 32 bit ray masks for geometry masking. On the CPU using any of the 32 bits yields the same performance, but the ray tracing hardware only supports an 8 bit mask, thus Embree has to emulate 32 bit masking if used. For that reason the lower 7 mask bits are hardware accelerated and fast, while the mask bits 7-31 require some software intervention and using them reduces performance. To turn on 32 bit ray masks use the RTC_FEATURE_FLAG_32_BIT_RAY_MASK (see section RTCFeatureFlags).

9.5 Limit Motion Blur Motions

The motion blur implementation on SYCL has some limitations regarding supported motion. Primitive motion should be maximally as large as a small multiple of the primitive size, otherwise performance can degrade a lot. If detailed geometry moves fast, best put the geometry into an instance, and apply motion blur to the instance itself, which efficiently allows larger motions. As a fallback, problematic scenes can always still get rendered robustly on the CPU.

9.6 Generic Pointers

Embree uses standard C++ pointers in its implementation. SYCL might not be able to detect the memory space these pointers refer to and has to treat them as generic pointers which are not performing optimal. The DPC++ compiler has advanced optimizations to infer the proper address space to avoid usage of generic pointers.

However, if you still encounter the following warning during ahead of time compilation of SYCL kernels, then loads from generic pointer are present:

```
warning: Adding XX occurrences of additional control flow due to presence
         of generic address space operations in function YYY.
```

To work around this issue we recommend:

- Do not use local memory inside kernels that trace rays. In this case the DPC++ compiler knows that no local memory pointer can exist and will optimize generic loads. As this is typically the case for renderers, generic pointer will typically not cause issues.
- Indirectly callable functions may still cause problems, even if your kernel does not use local memory. Thus best use SYCL pointers like sycl::global_ptr and sycl::private_ptr in indirectly callable functions to avoid generic address space usage.
- You can also enforce usage of global pointers using the following DPC++ compile flags: `-cl-intel-force-global-mem-allocation -cl-intel-no-local-to-generic`.

Chapter 10
Embree Tutorials

Embree comes with a set of tutorials aimed at helping users understand how Embree can be used and extended. There is a very basic minimal that can be compiled as both C and C++, which should get new users started quickly. All other tutorials exist in an Intel® ISPC and C++ version to demonstrate the two versions of the API. Look for files named `tutorialname_device.ispc` for the Intel® 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 Intel® ISPC version use the `tutorialname_ispc` executables. All tutorials can print available command line options using the `--help` command line parameter.

For all tutorials except minimal, 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 window size using the `--size` command line parameter, or start the tutorials in full screen using the `--fullscreen` parameter:

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

The initialization string for the Embree device (`rtcNewDevice` call) can be passed to the ray tracing core through the `--rtcore` command line parameter, e.g.:

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

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 Traces occlusion rays only.
F4 UV Coordinate visualization
F5 Geometry normal visualization
F6 Geometry ID visualization
F7 Geometry ID and Primitive ID visualization

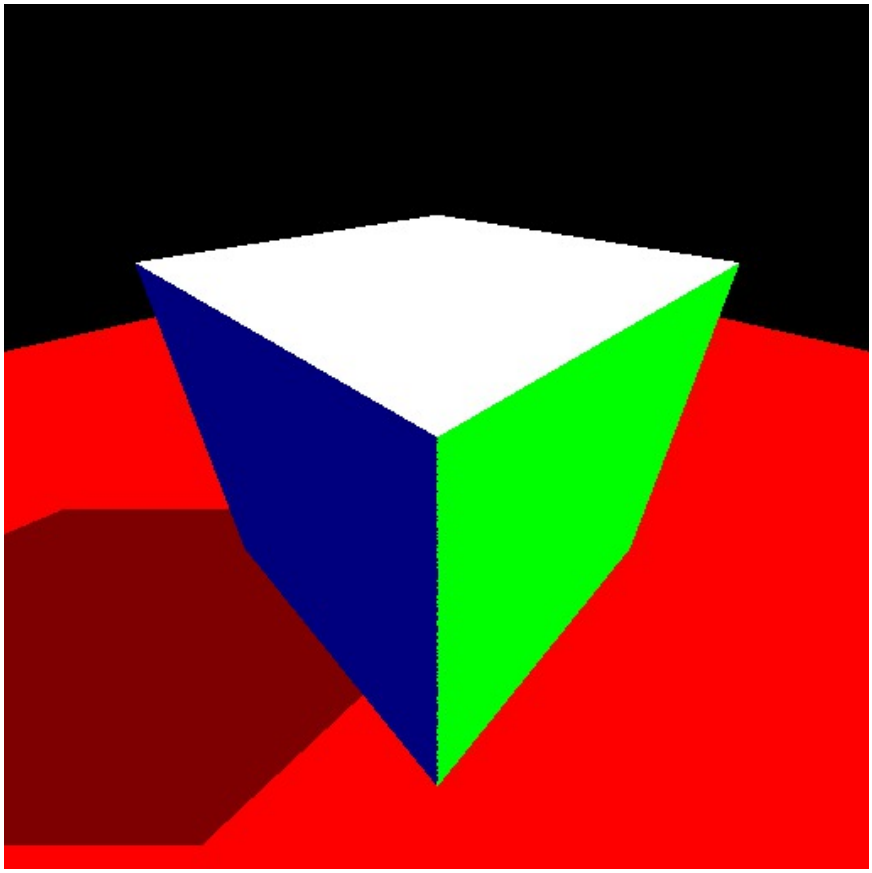
F8 Simple shading with 16 rays per pixel for benchmarking.
F9 Switches to render cost visualization. Pressing again reduces brightness.
F10 Switches to render cost visualization. Pressing again increases brightness.
f Enters or leaves full screen mode.
c Prints camera parameters.
ESC Exits the tutorial.
q Exits the tutorial.

10.1 Minimal

This tutorial is designed to get new users started with Embree. It can be compiled as both C and C++. It demonstrates how to initialize a device and scene, and how to intersect rays with the scene. There is no image output to keep the tutorial as simple as possible.

Source Code

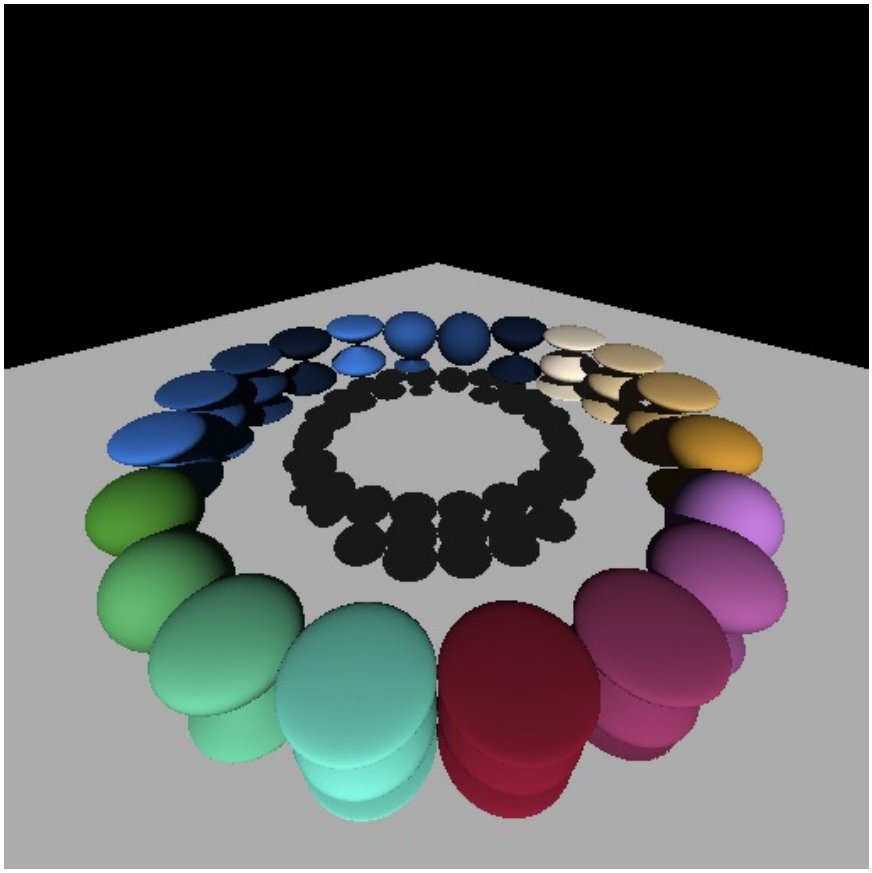
10.2 Triangle Geometry



This tutorial demonstrates the creation of a static cube and ground plane using triangle meshes. It also demonstrates the use of the `rtcIntersect1` and `rtcOccluded1` functions to render primary visibility and hard shadows. The cube sides are colored based on the ID of the hit primitive.

Source Code

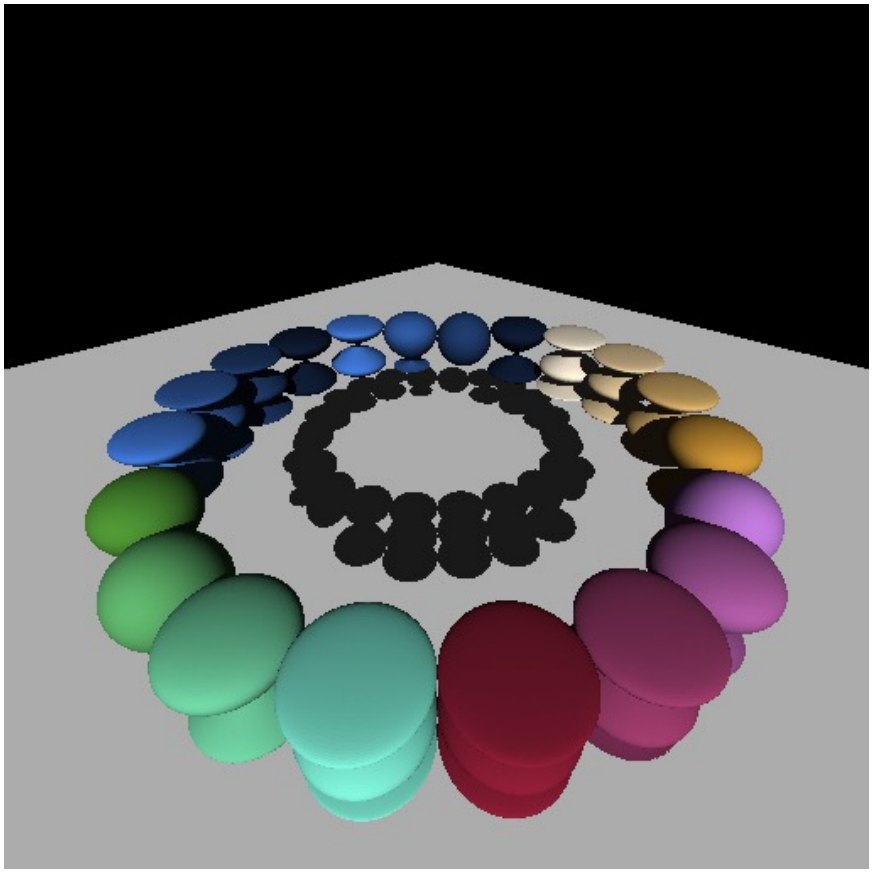
10.3 Dynamic Scene



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

Source Code

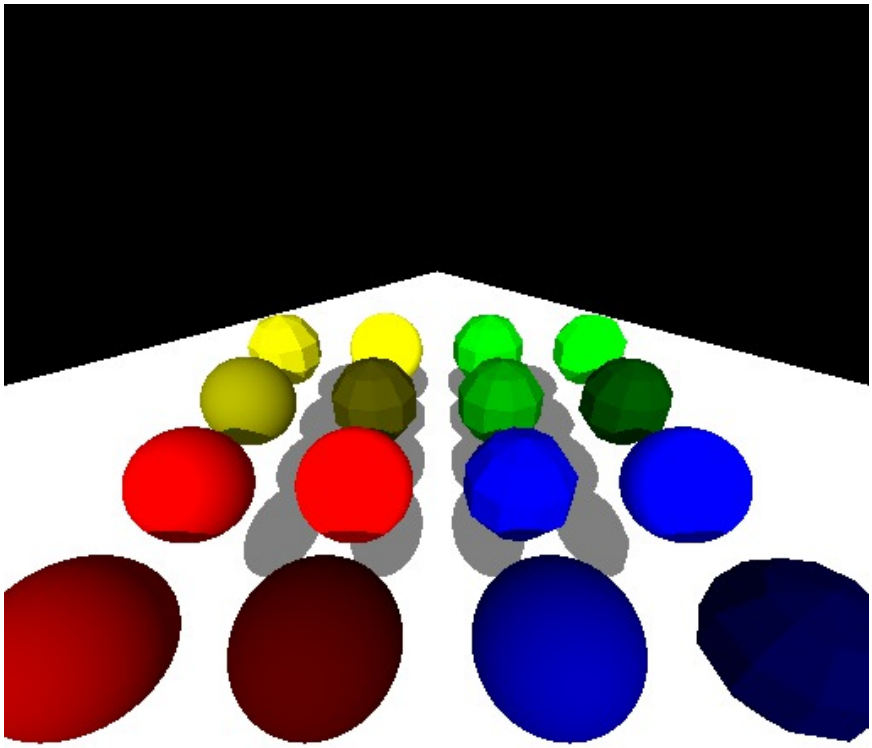
10.4 Multi Scene Geometry



This tutorial demonstrates the creation of multiple scenes sharing the same geometry objects. Here, three scenes are built. One with all the dynamic spheres of the Dynamic Scene test and two others each with half. The ground plane is shared by all three scenes. The space bar is used to cycle the scene chosen for rendering.

Source Code

10.5 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 kinds. 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.

Source Code

10.6 Viewer

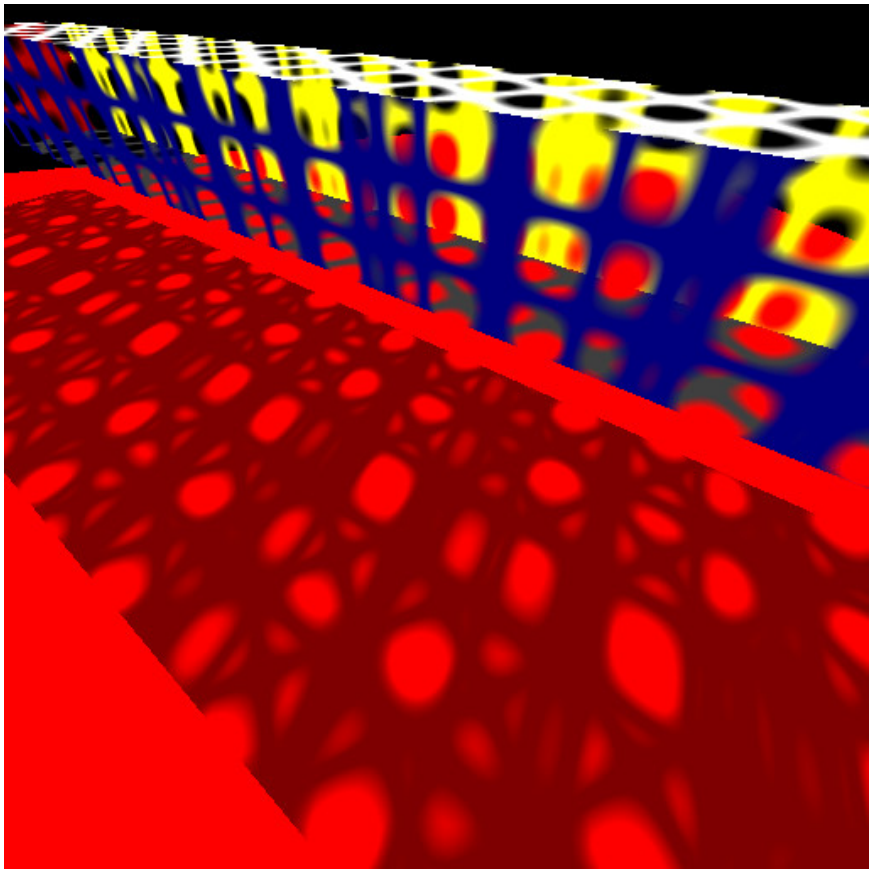
This tutorial demonstrates a simple OBJ viewer that traces primary visibility rays only. A scene consisting of multiple meshes is created, each mesh sharing the index and vertex buffer with the application. It also demonstrates how to support additional per-vertex data, such as shading normals.

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

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

Source Code

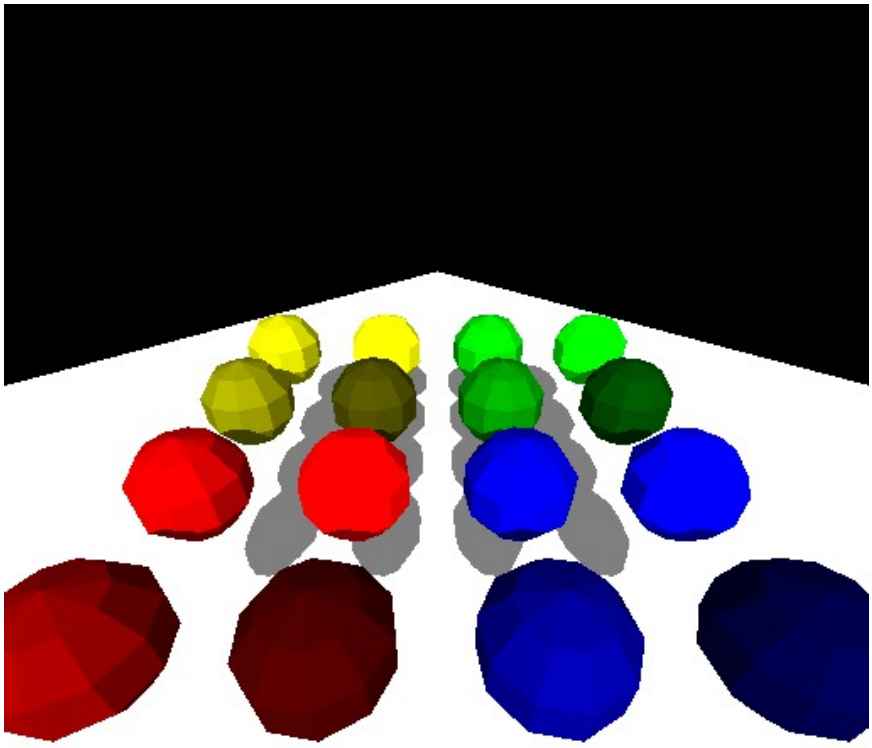
10.7 Intersection Filter



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.

Source Code

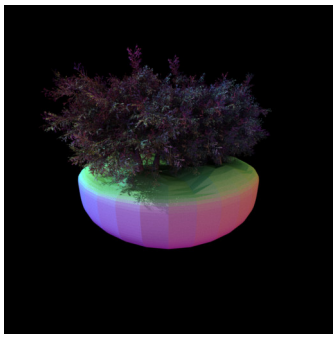
10.8 Instanced Geometry



This tutorial demonstrates the in-build instancing feature of Embree, by instancing a number of other scenes built 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.

Source Code

10.9 Multi Level Instancing



This tutorial demonstrates multi-level instancing, i.e., nesting instances into instances. To enable the tutorial, set the compile-time variable `EMBREE_MAX_INSTANCE_LEVEL_COUNT` to a value other than the default 1. This variable is available in the code as `RTC_MAX_INSTANCE_LEVEL_COUNT`.

The renderer uses a basic path tracing approach, and the image will progressively refine over time. There are two levels of instances in this scene: mul-

tiple instances of the same tree nest instances of a twig. Intersections on up to `RTC_MAX_INSTANCE_LEVEL_COUNT` nested levels of instances work out of the box. Users may obtain the *instance ID stack* for a given hitpoint from the `instID` member. During shading, the instance ID stack is used to accumulate normal transformation matrices for each hit. The tutorial visualizes transformed normals as colors.

Source Code

10.10 Path Tracer

This tutorial is a simple path tracer, based 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:

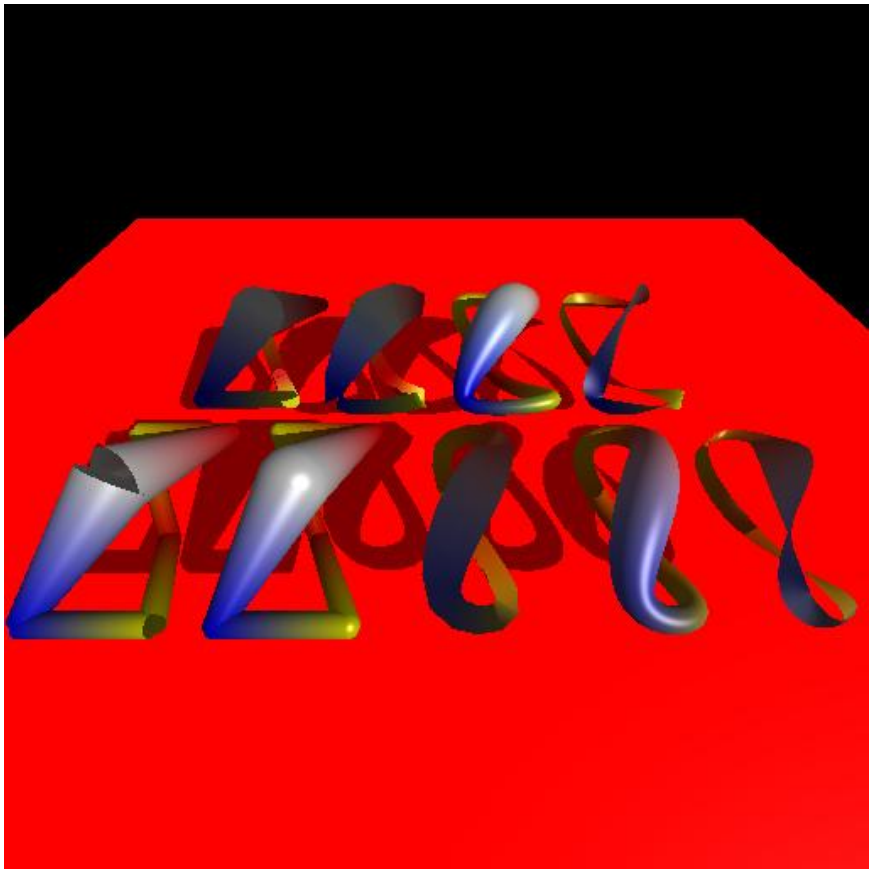
```
./pathtracer -c crown/crown.ecs
./pathtracer -c asian_dragon/asian_dragon.ecs
```

Source Code

10.11 Hair

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

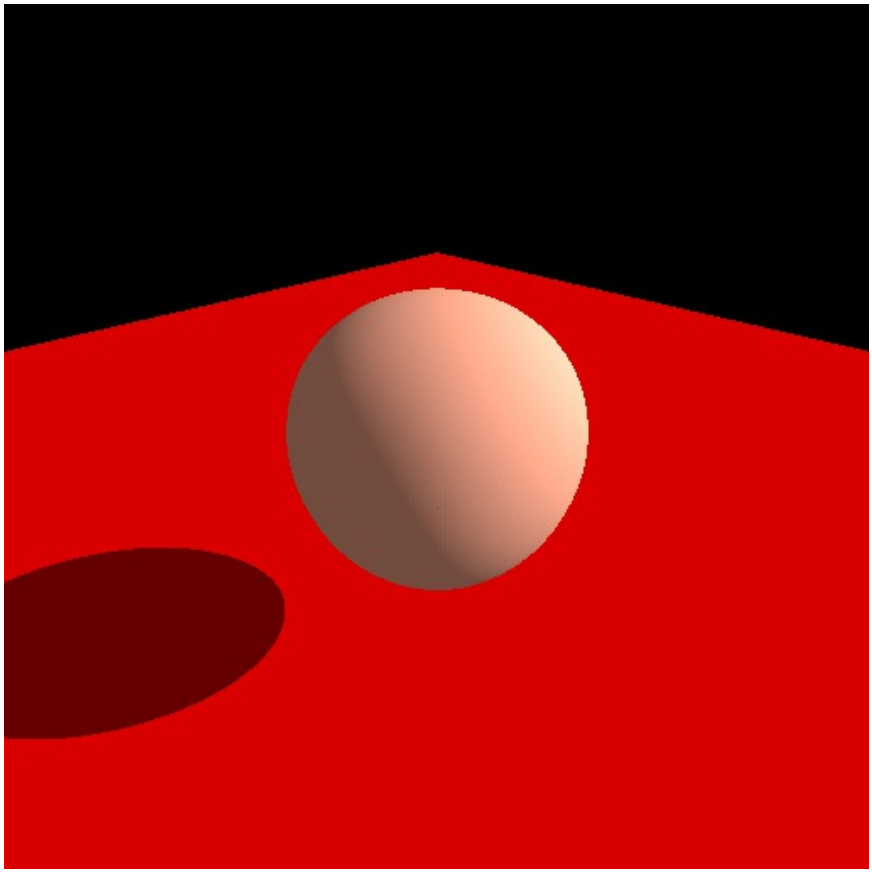
10.12 Curve Geometry



This tutorial demonstrates the use of the Linear Basis, B-Spline, and Catmull-Rom curve geometries.

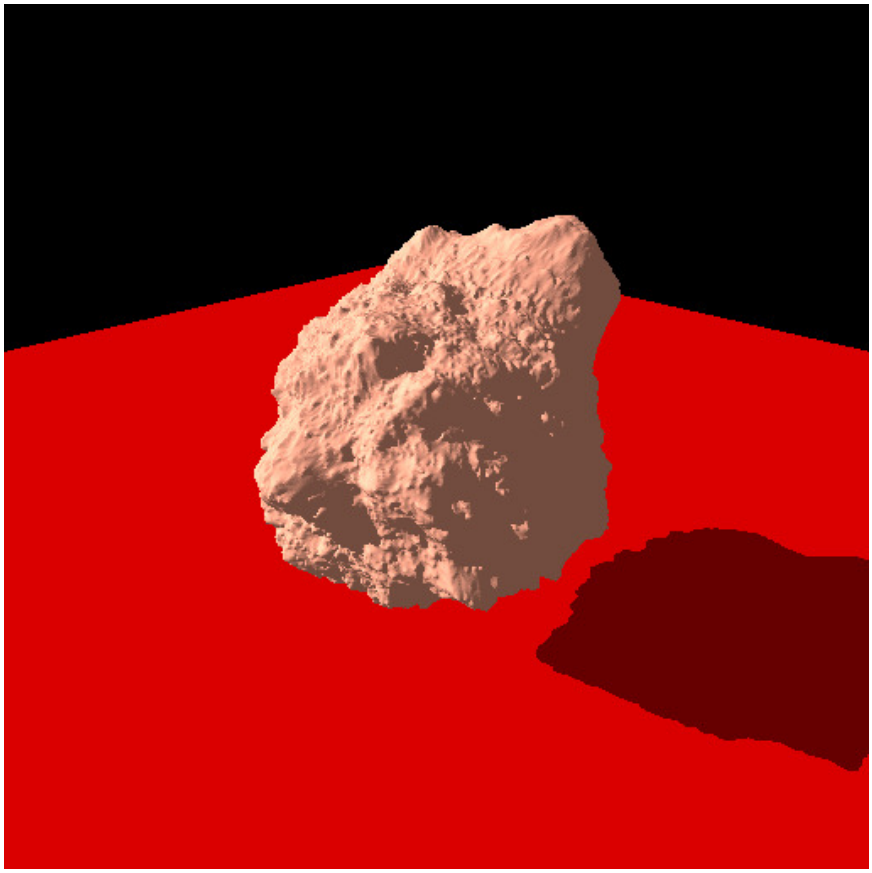
Source Code

10.13 Subdivision Geometry



This tutorial demonstrates the use of Catmull-Clark subdivision surfaces.
Source Code

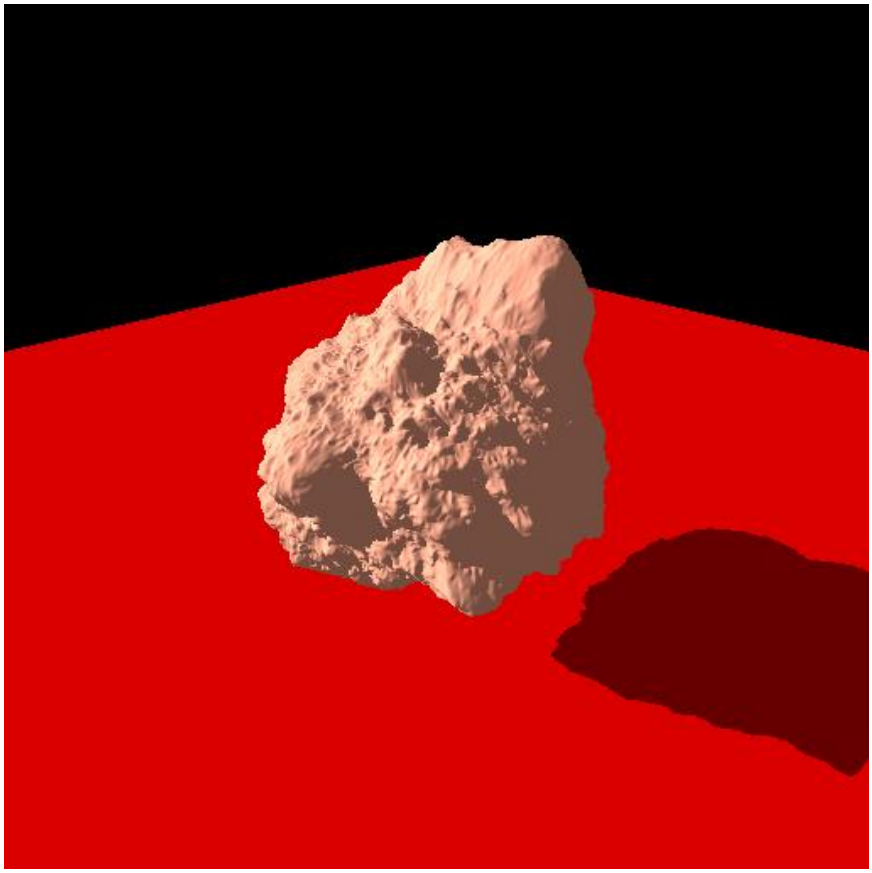
10.14 Displacement Geometry



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

Source Code

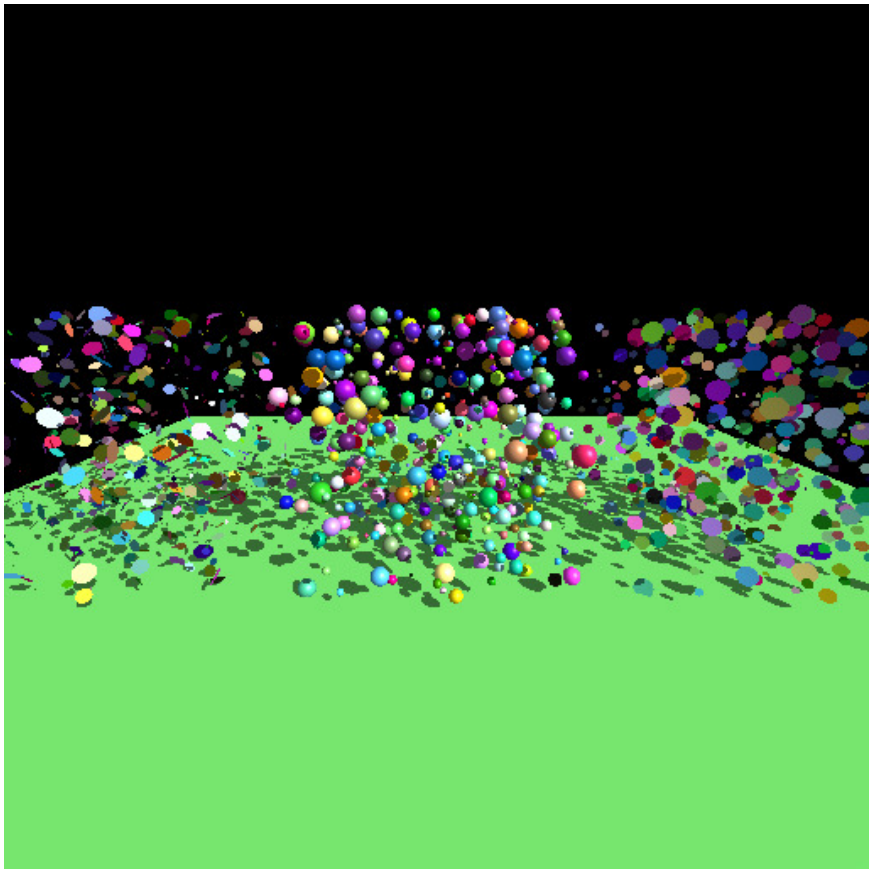
10.15 Grid Geometry



This tutorial demonstrates the use of the memory efficient grid primitive to handle highly tessellated and displaced geometry.

Source Code

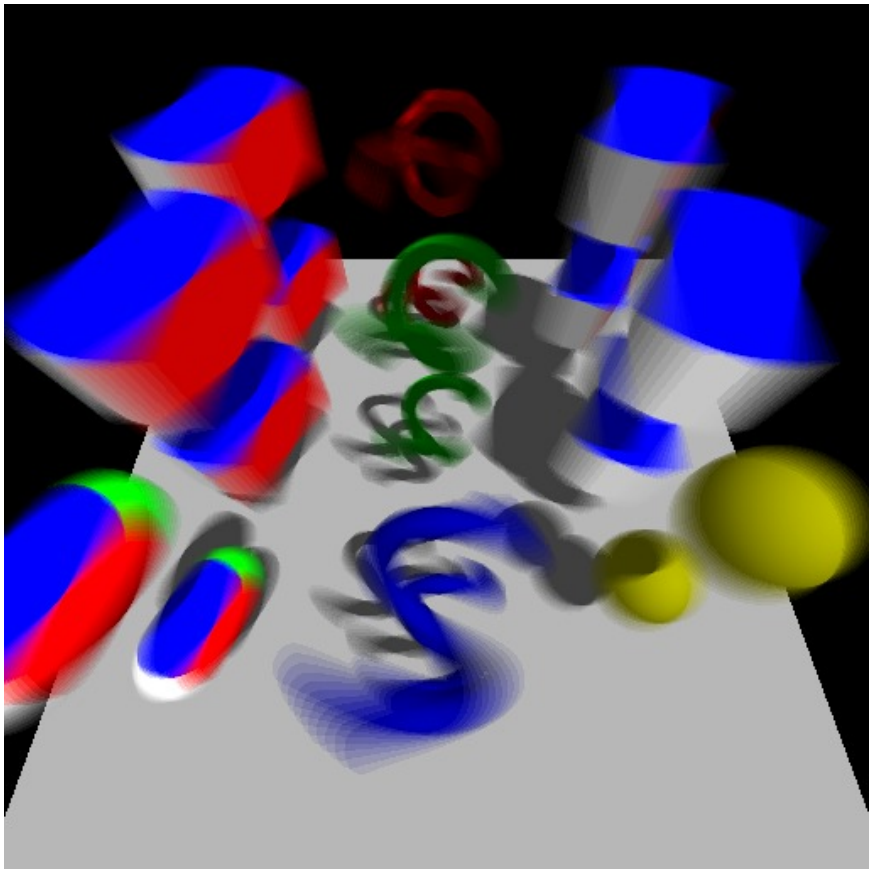
10.16 Point Geometry



This tutorial demonstrates the use of the three representations of point geometry.

Source Code

10.17 Motion Blur Geometry

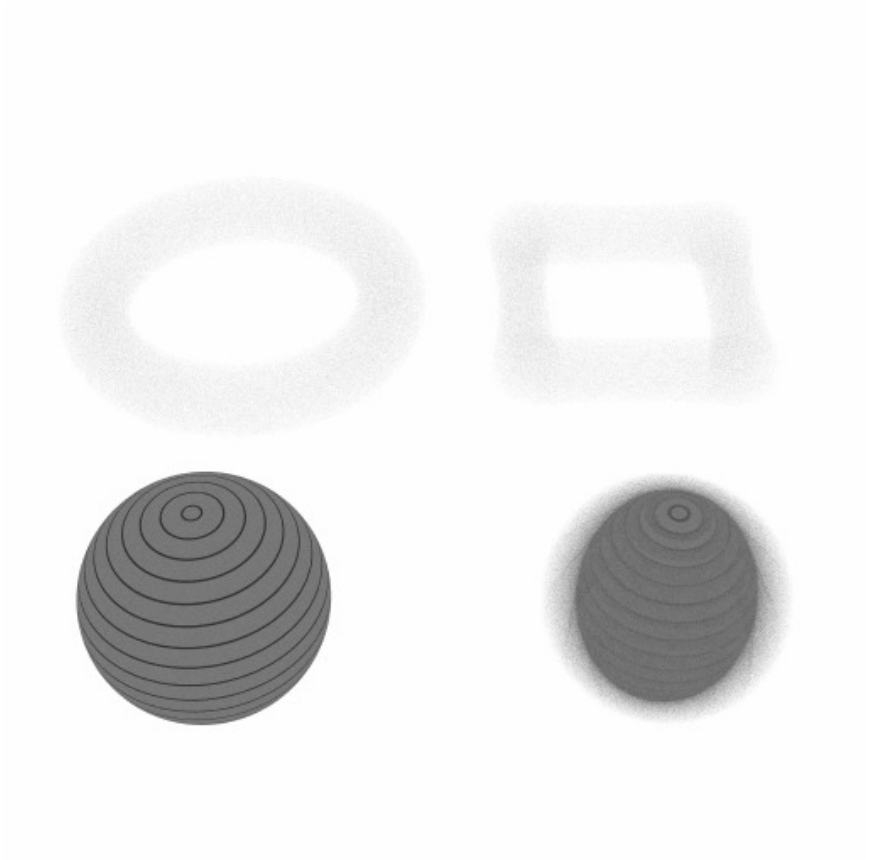


This tutorial demonstrates rendering of motion blur using the multi-segment motion blur feature. Shown is motion blur of a triangle mesh, quad mesh, subdivision surface, line segments, hair geometry, Bézier curves, instantiated triangle mesh where the instance moves, instantiated quad mesh where the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

Source Code

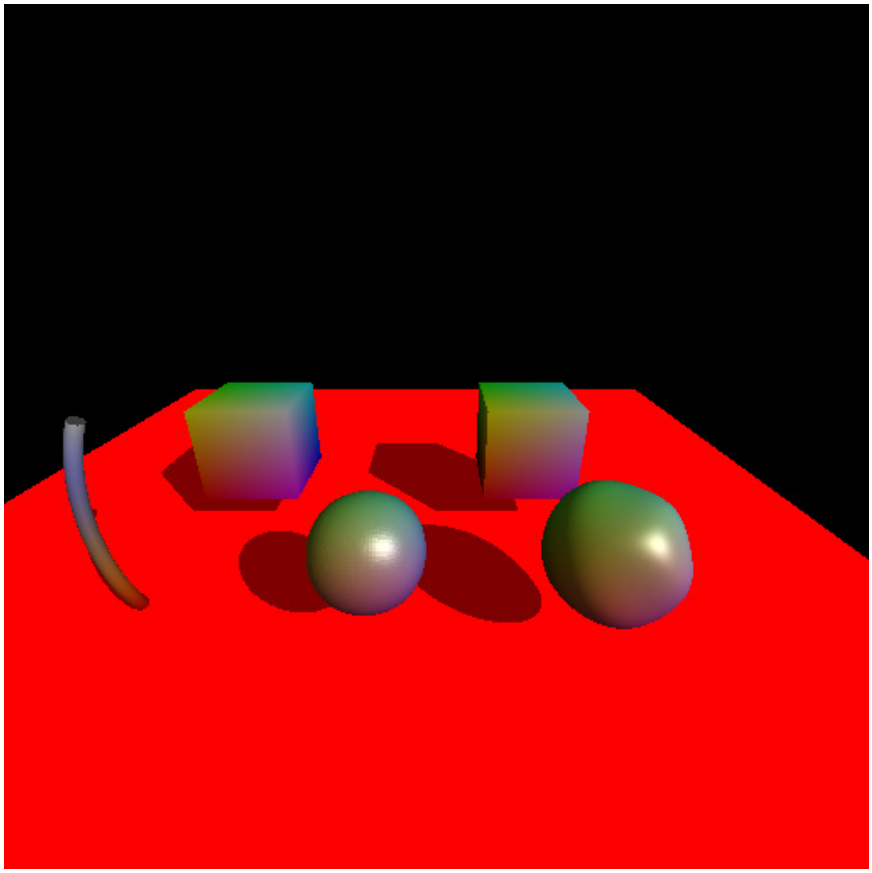
10.18 Quaternion Motion Blur



This tutorial demonstrates rendering of motion blur using quaternion interpolation. Shown is motion blur using spherical linear interpolation of the rotational component of the instance transformation on the left and simple linear interpolation of the instance transformation on the right. The number of time steps can be modified as well.

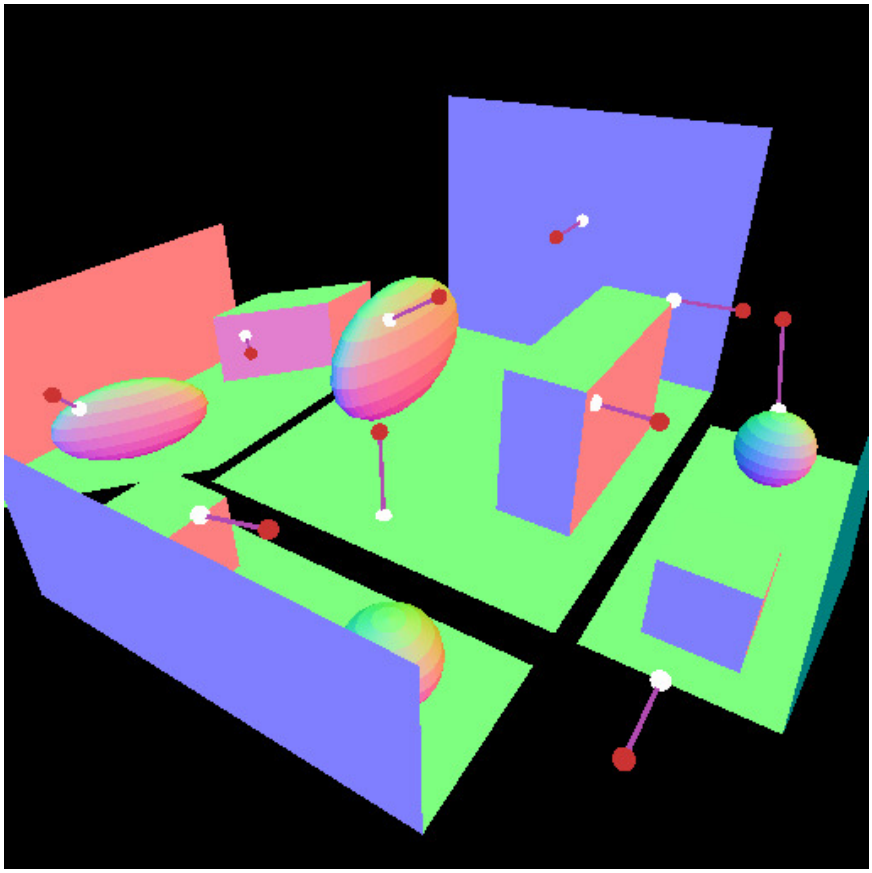
Source Code

10.19 Interpolation



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

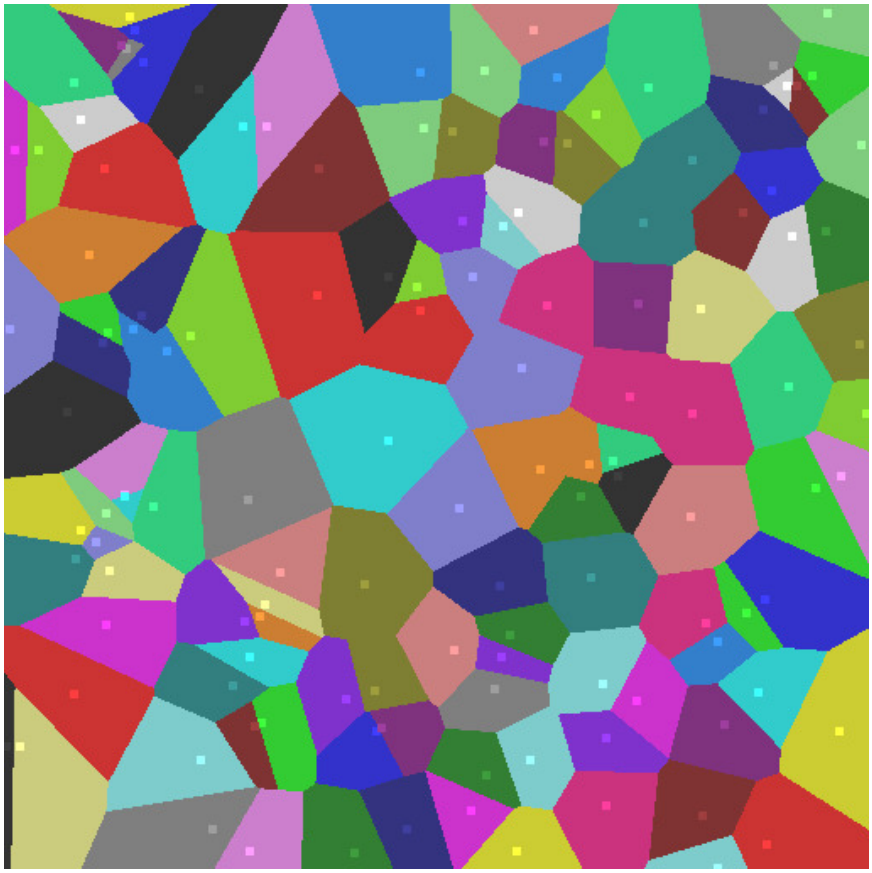
10.20 Closest Point



This tutorial demonstrates a use-case of the point query API. The scene consists of a simple collection of objects that are instanced and for several point in the scene (red points) the closest point on the surfaces of the scene are computed (white points). The closest point functionality is implemented for Embree internal and for user-defined instancing. The tutorial also illustrates how to handle instance transformations that are not similarity transforms.

Source Code

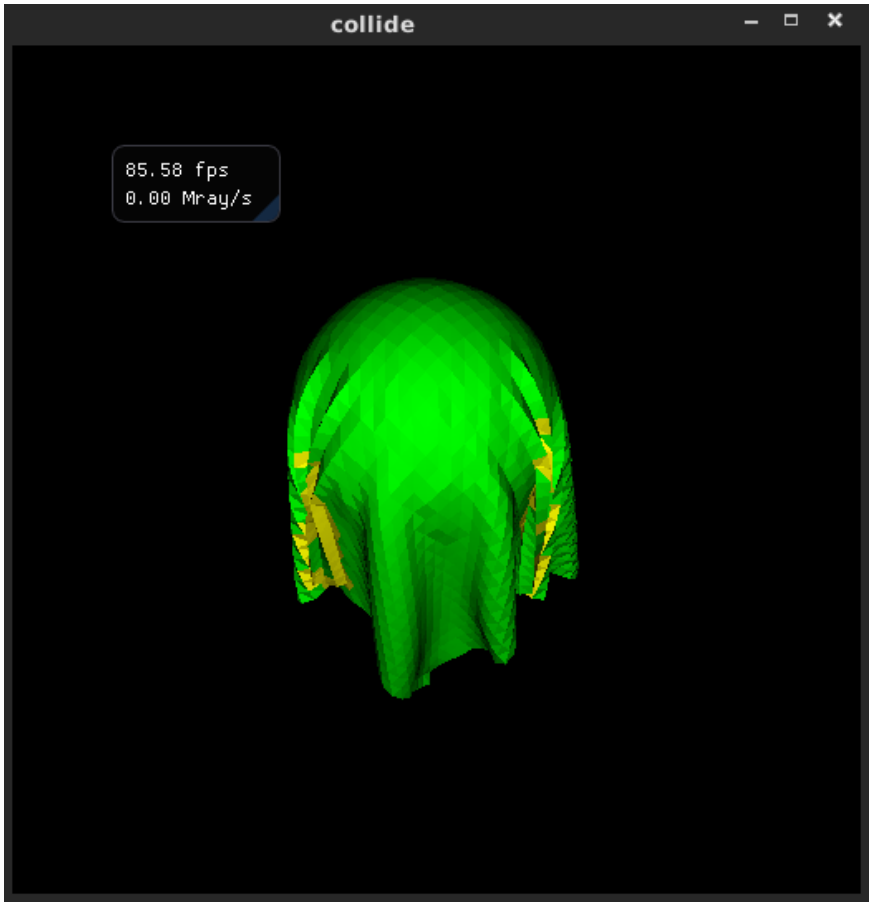
10.21 Voronoi



This tutorial demonstrates how to implement nearest neighbour lookups using the point query API. Several colored points are located on a plane and the corresponding voroni regions are illustrated.

Source Code

10.22 Collision Detection



This tutorial demonstrates how to implement collision detection using the collide API. A simple cloth solver is setup to collide with a sphere.

The cloth can be reset with the `space` bar. The sim stepped once with `n` and continuous simulation started and paused with `p`.

Source Code

10.23 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 using spatial splits, a standard SAH builder, and a very fast Morton builder.

Source Code

10.24 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.

Source Code

10.25 Find Embree

This tutorial demonstrates how to use the `FIND_PACKAGE` CMake feature to use an installed Embree. Under Linux and macOS the tutorial finds the Embree installation automatically, under Windows the `embree_DIR` CMake variable must be set to the following folder of the Embree installation: `C:\Program Files\Intel\Embree3`.

Source Code

10.26 Next Hit

This tutorial demonstrates how to robustly enumerate all hits along the ray using multiple ray queries and an intersection filter function. To improve performance, the tutorial also supports collecting the next N hits in a single ray query.

Source Code

© 2009–2020 Intel Corporation

Intel, the Intel logo, Xeon, Intel Xeon Phi, and Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Intel optimizations, for Intel compilers or other products, may not optimize to the same degree for non-Intel products.