Embree API

The Embree API is a low level ray tracing API that supports defining and committing of geometry and performing ray queries of different types. Static and dynamic scenes are supported, that may contain triangular geometry (including linear motions for motion blur), instanced geometry, and user defined geometry. Supported ray queries are, finding the closest scene intersection along a ray, and testing a ray segment for any intersection with the scene. Single rays, as well as packets of rays in a struct of array layout can be used for packet sizes of 1, 4, 8, and 16.

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

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

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

All API calls carry the prefix `rtc` which stands for **r**ay **t**racing **c**ore. Before invoking any API call, the Embree ray tracing core has to get initialized through the `rtcInit` call. Before the application exits it should call `rtcExit`. Initializing Embree again after an `rtcExit` is allowed.

```
rtcInit(NULL);
...
rtcExit();
```

The `rtcInit` call initializes the ray tracing core. An optional configuration string can be passed through this function to configure implementation specific parameters. If this string is NULL, a default configuration is used, that is optimal for most usages.

API calls that access geometries are only thread safe as long as different geometries are accessed. Accesses to one geometry have to get sequentialized by the application. All other API calls are thread safe. The `rtcIntersect` and `rtcOccluded` calls are re-entrant, but only for other `rtcIntersect` and `rtcOccluded` calls. It is thus safe to trace new rays when intersecting a user defined object, but not supported to create new geometry inside the intersect function of a user defined geometry.

Each user thread has its own error flag in the API. If an error occurs when invoking some API function, this flag is set to an error code if it stores no previous error. The `rtcGetError` function reads and returns the currently stored error and clears the error flag again. For performance reasons the ray query functions do not set an error flag, but fail hard in some circumstances (e.g. if the user forgot to perform an `rtcCommit`).

Possible error codes returned by `rtcGetError` are:

Error Code	Description
RTC_NO_ERROR	No error occured.
RTC_UNKNOWN_ERROR	An unknown error has occured.
RTC_INVALID_ARGUMENT	An invalid argument was specified.
RTC_INVALID_OPERATION	The operation is not allowed for the specified object.

RTC_OUT_OF_MEMORY	There is not enough memory left to
RTC_UNSUPPORTED_CPU	The CPU is not supported as it does not support SSE2.

Scene

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

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

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

Scene Flag	Description
RTC_SCENE_STATIC	scene optimized for static geometry
RTC_SCENE_DYNAMIC	scene optimized for dynamic geometry

A dynamic scene is created by invoking `rtcNewScene` with the `RTC_SCENE_DYNAMIC` flag. Different geometries can now be created inside that scene. Geometries are enabled by default. Once the scene geometry is specified, an `rtcCommit` call will finish the scene description and trigger building of internal data structures. After the `rtcCommit` call it is safe to perform ray queries of the type specified at scene construction time. Geometries can get disabled (`rtcDisable` call), enabled again (`rtcEnable` call), and deleted (`rtcDeleteGeometry` call). Geometries can also get modified, including their vertex and index arrays. After the modification of some geometry, `rtcModified` has to get called for that geometry. If geometries got enabled, disabled, deleted, or modified an `rtcCommit` call has to get invoked before performing any ray queries for the scene, otherwise the effect of the ray query is undefined.

A static scene is created by the `rtcNewScene` call with the `RTC_SCENE_STATIC` flag. Geometries can only be created and modified until the first `rtcCommit` call. After the `rtcCommit` call, each access to any geometry of that static scene is invalid, including enabling, disabling, modifying, and deletion of geometries. Consequently, geometries that got created inside a static scene can only get deleted by deleting the entire scene.

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

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

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

Scene Flag	Description
RTC_SCENE_ROBUST	Avoid optimizations that reduce arithmetic accuracy.

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

Algorithm Flag	Description
RTC_INTERSECT1	Enables the `rtcIntersect` and `rtcOccluded` functions (single ray interface) for this scene
RTC_INTERSECT4	Enables the `rtcIntersect4` and `rtcOccluded4` functions (4-wide packet interface) for this scene
RTC_INTERSECT8	Enables the `rtcIntersect8` and `rtcOccluded8` functions (8-wide packet interface) for this scene
RTC_INTERSECT16	Enables the `rtcIntersect16` and `rtcOccluded16` functions (16-wide packet interface) for this scene

Geometries

Geometries are always contained in the scene they are created in. Each geometry is assigned an integer ID at creation time, which is unique for that scene. The current version of the API supports triangle meshes (`rtcNewTriangleMesh`), single level instances of other scenes (`rtcNewInstance`), and user defined geometries (`rtcNewUserGeometry`). The API is designed in a way that easily allows adding new geometry types in later releases.

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

For static scenes, geometry IDs are assigned sequentially starting at 0. This allows the application to use a fixed size array to map from geometry IDs to its own geometry representation.

Triangle Meshes

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

The number of triangles, number of vertices, and number of time steps (1 for normal meshes, and 2 for linear motion blur), have to get specified at construction time of the mesh. The user can also specify additional flags that choose the strategy to handle that mesh in dynamic scenes. The following example demonstrates howto create a triangle mesh without motion blur:

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

The following geometry flags can be specified at construction time of the triangle mesh:

Geometry Flag	Description
RTC_GEOMETRY_STATIC	The mesh is considered static and should get modified rarely by the application. This flag has to get used in static scenes.
RTC_GEOMETRY_DEFORMABLE	The mesh is considered to deform in a coherent way, e.g. a skinned character. The connectivity of the mesh has to stay constant, thus modifying the index array is not allowed. The implementation is free to choose a BVH refitting

	approach for handling meshes tagged with that flag.
RTC_GEOMETRY_DYNAMIC	The mesh is considered highly dynamic and changes frequently, possibly in an unstructured way. Embree will rebuild data structures from scratch for this type of mesh.

The triangle indices can be set by mapping and writing to the index buffer (`RTC_INDEX_BUFFER`) and the triangle vertices can be set by mapping and writing into the vertex buffer (`RTC_VERTEX_BUFFER`). The index buffer contains an array of three 32 bit indices, while the vertex buffer contains an array of 3 float values aligned to 16 bytes. All buffers have to get unmapped before an `rtcCommit` call to the scene.

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

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

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

A triangle mesh with linear motion blur support is created by setting the number of time steps to 2 at mesh construction time. Specifying a number of time steps of 0 or larger than 2 is invalid. For a triangle mesh with linear motion blur, the user has to set the `RTC_VERTEX_BUFFER0` and `RTC_VERTEX_BUFFER1` vertex arrays, one for each time step. If a scene contains triangle meshes with linear motion blur, the user has to set the `time` member of the ray to a value in the range [0,1]. The ray will intersect the scene with the vertices of the two time steps linearly interpolated to this specified time. Each ray can specify a different time, even inside a ray packet.

A 30 bit geometry mask can be assigned to triangle mesh geometries using the `rtcSetMask` call.

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

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

See tutorial00 for an example of how to create triangle meshes.

User Defined Geometry

User defined geometries make it possible to extend Embree with arbitrary types of geometry. This is achieved by introducing arrays of user geometries as a special geometry type. These objects do not contain a single user geometry, but a set of such geometries, each specified by an index. The user has to provide a user data pointer, bounding function as well as user defined intersect and occluded functions to create a set of user geometries. The user geometry to process is specified by passing its user data pointer and index to each invokation of the bounding, intersect, and occluded function. The bounding function is used to query the bounds of each user geometry. When performing ray queries, Embree will invoke the user intersect (and occluded) functions to test rays for intersection (and occlusion) with the specified user defined geometry.

As Embree supports different ray packet sizes, one potentially has to provide different versions of user intersect and occluded function pointers for these packet sizes. However, the ray packet size of the called user function always matches the packet size of the originally invoked ray query function. Consequently, an application only operating on single rays only has to provide single ray intersect and occluded function pointers.

User geometries are created using the `rtcNewUserGeometry` function call, and potentially deleted using the `rtcDeleteGeometry` function call. The folling example illustrates creating an array with two user geometries:

```
struct UserObject { ... };

void userBoundsFunction(UserObject* userGeom, size_t i, RTCBounds& bounds_o) {
  bounds_o = bounds of userGeom[i];
}

void userIntersectFunction(UserObject* userGeom, RTCRay& ray, size_t i) {
  if (ray misses userGeom[i]) return;
  update ray hit information;
}

void userOccludedFunction(UserObject* userGeom, RTCRay& ray, size_t i) {
  if (ray misses userGeom[i]) return;
  geomID = 0;
}

...

UserObject* userGeom = new UserObject[2];
userGeom[0] = ...
userGeom[1] = ...
unsigned geomID = rtcNewUserGeometry(scene,2);
rtcSetUserData(scene,geomID,userGeom);
rtcSetBounds(scene,geomID,userBoundsFunction);
rtcSetIntersectFunction(scene,geomID,userIntersectFunction);
rtcSetOccludedFunction (scene,geomID,userOccludedFunction);
```

The user intersect function (`userIntersectFunction`) and user occluded function (`userOccludedFunction`) get as input the pointer provided through the `rtcSetUserData` function call, a ray, and the index of the geometry to process. For ray packets, the user intersect and occluded functions also get a pointer to a valid mask as input. The user provided functions should not modify any ray that is disabled by that valid mask.

The user intersect function should return without modifying the ray structure if the user geometry is missed. If the geometry is hit, it has to update the hit information of the ray (`tfar`, `u`, `v`, `Ng`, `geomID`, `primID`).

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

Is is supported to invoke the `rtcIntersect` and `rtcOccluded` function calls inside such user functions. It is not supported to invoke any other API call inside these user functions.

See tutorial02 for an example of how to use the user defined geometries.

Instances

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

Instances are created using the `rtcNewInstance` function call, and potentially deleted using the `rtcDeleteGeometry` function call. To instantiate a scene, one first has to generate the scene B to instantiate. Now one can add an instance of this scene inside a scene A the following way:

```
unsigned instID = rtcNewInstance(sceneA,sceneB);
rtcSetTransform(sceneA,instID,RTC_MATRIX_COLUMN_MAJOR,&column_matrix_3x4);
```

One has to call `rtcCommit` on scene B before one calls `rtcCommit` on scene A. When modifying scene B one has to call `rtcModified` for all instances of that scene. Providing a bounding box is not required and also not allowed. If a ray hits the instance, then the geomID and primID members of the ray are set to the geometry ID and primitive ID of the primitive hit in scene B, and the instID member of the ray is set to the instance ID returned from the `rtcNewInstance` function.

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

Layout	Description
RTC_MATRIX_ROW_MAJOR	The 3x4 float matrix is layed out in row major form.
RTC_MATRIX_COLUMN_MAJOR	The 3x4 float matrix is layed out in column major form.
RTC_MATRIX_COLUMN_MAJOR_ALIGNED16	The 3x4 float matrix is layout out in column major form, with each column padded by an additional 4th component.

Passing homogenous 4x4 matrices is possible as long as the last row is (0,0,0,1). If this homogenous matrix is layed out in row major form, use the `RTC_MATRIX_ROW_MAJOR` layout. If this homogenous matrix is layed out in column major form, use the `RTC_MATRIX_COLUMN_MAJOR_ALIGNED16` mode. In both cases, Embree will ignore the last row of the matrix.

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

See tutorial04 for an example of how to use instances.

Ray Queries

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

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

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

Member	In/Out	Description
org	in	ray origin
dir	in	ray direction (can be unnomalized)
tnear	in	start of ray segment
tfar	in/out	end of ray segment, set to hit distance after intersection

time	in	time used for motion blur
mask	in	ray mask to mask out geometries (30 bits used)
Ng	out	not normalized geometry normal
u	out	barycentric u-coordinate of hit
v	out	barycentric v-coordinate of hit
geomID	out	geometry ID of hit geometry
primID	out	primitive ID of hit primitive
instID	out	instance ID of hit instance

This structure is in struct of array layout (SOA) for ray packets. Note that the `tfar` member functions as an input and output.

In the ray packet mode (with packet size of N), the user has to provide a pointer to N 32 bit integers that act as a ray activity mask. If one of these integers is set to 0x00000000 the corresponding ray is considered inactive and if the integer is set to 0xFFFFFFFF, the ray is considered active. Rays that are inactive will not update any hit information. Data alignment requirements for ray query functions operating on single rays is 16 bytes for the ray.

Data alignment requirements for query functions operating on AOS packets of 4, 8, or 16 rays, is 16, 32, and 64 bytes respectively, for the valid mask and the ray. To operate on packets of 4 rays, the CPU has to support SSE, to operate on packets of 8 rays, the CPU has to support AVX-256, and to operate on packets of 16 rays, the CPU has to support the Xeon Phi instructions. Additionally, the required ISA has to be enabled in Embree at compile time, to use the desired packet size.

Finding the closest hit distance is done through the `rtcIntersect` functions. These get the activity mask, the scene, and a ray as input. The user has to initialize the ray origin (`org`), ray direction (`dir`), and ray segment (`tnear`, `tfar`). The ray segment has to be in the range [0,inf], thus ranges that start behind the ray origin are not valid, but ranges can reach to infinity. The geometry ID (`geomID` member) has to get initialized to `RTC_INVALID_GEOMETRY_ID` (-1). If the scene contains instances, also the instance ID (`instID`) has to get initialized to `RTC_INVALID_GEOMETRY_ID` (-1). If the scene contains linear motion blur, also the ray time (`time`) has to get initialized to a value in the range [0,1]. If ray masks are enabled at compile time, also the ray mask (`mask`) has to get initialized. After tracing the ray, the hit distance (`tfar`), geometry normal (`Ng`), local hit coordinates (`u`, `v`), geometry ID (`geomID`), and primitive ID (`primID`) are set. If the scene contains instances, also the instance ID (`instID`) is set, if an instance is hit. The geometry ID corresponds to the ID returned at creation time of the hit geometry, and the primitive ID corresponds to the nth primitive of that geometry, e.g. nth triangle. The instance ID corresponds to the ID returned at creation time of the instance.

The following code properly sets up a ray and traces it through the scene:

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

Testing if any geometry intersects with the ray segment is done through the `rtcOccluded` functions. Initialization has

to be done as for `rtcIntersect`. If some geometry got found along the ray segment, the geometry ID (`geomID`) will get set to 0. No other member of the ray will get modified.

See tutorial00 for an example of how to trace rays.

Filter Functions

The API supports per geometry filter callback functions that are invoked for each intersection found during the `rtcIntersect` or `rtcOccluded` calls. The former ones are called intersection filter functions, the latter ones occlusion filter functions. The filter functions can be used to implement various useful features, such as rejecting a hit to implement backface culling, accumulating opacity for shadow shadows, counting the number of surfaces along a ray, collecting all hits along a ray, etc. The filter functions are only supported for triangle mesh geometry, including triangle meshes with motion blur.

The filter functions provided by the user have to have the following signature:

```
void FilterFunc  (                  void* userPtr, RTCRay&   ray);
void FilterFunc4 (const void* valid, void* userPtr, RTCRay4&  ray);
void FilterFunc8 (const void* valid, void* userPtr, RTCRay8&  ray);
void FilterFunc16(const void* valid, void* userPtr, RTCRay16& ray);
```

The `valid` pointer points to a valid mask of the same format as expected as input by the ray query functions. The `userPtr` is a user pointer optionally set per geometry through the `rtcSetUserData` function. The ray passed to the filter function is the ray structure initially provided to the ray query function by the user. For that reason, it is safe to extend the ray by additional data and access this data inside the filter function (e.g. to accumulate opacity). All hit information inside the ray is valid. If the hit geometry is instanced, the `instID` member of the ray is valid and the ray origin, direction, and geometry normal visible through the ray are in object space. The filter function can reject a hit by setting the `geomID` member of the ray to `RTC_INVALID_GEOMETRY_ID`, otherwise the hit is accepted. The filter function is not allowed to modify the ray input data (`org`, `dir`, `tnear`, `tfar`), but can modify the hit data of the ray (`u`,`v`,`Ng`,`geomID`,`primID`).

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

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

These functions are invoked during execution of the `rtcIntersect` type queries of the matching ray type. The occlusion filter functions are set using the following API functions:

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

See tutorial05 for an example of how to use the filter functions.