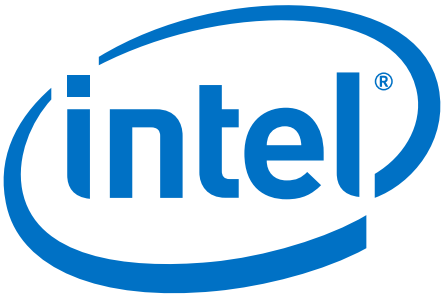


Embree
High Performance Ray Tracing Kernels

Version 2.4.0 (devel)
December 12, 2014

Contents

Chapter 1
Embree Overview

Embree is a collection of high-performance ray tracing kernels, developed at Intel. The target user of Embree are graphics application engineers that want to improve the performance of their application by leveraging the optimized ray tracing kernels of Embree. The kernels are optimized for photo-realistic rendering on the latest Intel® processors with support for SSE, AVX, AVX2, and the 16-wide Xeon Phi™ vector instructions. Embree supports runtime code selection to choose the traversal and build algorithms that best matches the instruction set of your CPU. We recommend using Embree through its API to get the highest benefit from future improvements. Embree is released as Open Source under the Apache 2.0 license.

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

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

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

1.1 Supported Platforms

Embree supports Windows (32 bit and 64 bit), Linux (64 bit) and Mac OS X (64 bit). The code compiles with the Intel Compiler, the Microsoft Compiler, GCC and CLANG. Using the Intel Compiler improves performance by approximately 10%. Performance also varies across different operating systems. Embree is optimized for Intel CPUs supporting SSE, AVX, and AVX2 instructions, and requires at least a CPU with support for SSE2.

The Xeon Phi™ version of Embree only works under Linux in 64 bit mode. For compilation of the the Xeon Phi™ code the Intel Compiler is required. The host side code compiles with GCC, CLANG, and the Intel Compiler.

1.2 Embree Support and Contact

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

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

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

For information about compiler optimizations, see our Optimization Notice.

Chapter 2
Compiling Embree

2.1 Linux and Mac OS X

Embree requires the Intel® SPMD Program Compiler (ISPC) to compile. We have tested ISPC version 1.7.1 and 1.8.0, but more recent versions of ISPC should also work. You can download and install the ISPC binaries from ispc.github.io. After installation, either put the path to the `ispc` executable permanently into your `PATH`:

```
export PATH=path-to-ispc:$PATH
```

Or provide the path to the `ispc` executable to CMake via the `ISPC_EXECUTABLE` variable.

You additionally have to install CMake 2.8.12 or higher and the developer version of GLUT. Under Mac OS X, these dependencies can be installed using MacPorts:

```
sudo port install cmake freeglut
```

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

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

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

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

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

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

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

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

```
make
```

The executables will be generated inside the build folder. We recommend to finally install the Embree library and header files on your system:

```
sudo make install
```

If you cannot install Embree on your system (e.g. when you don't have administrator rights) you need to add embree_root_directory/build to your `LD_LIBRARY_PATH` (and `SINK_LD_LIBRARY_PATH` in case you want to use Embree on Xeon Phi™).

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

Table 2.1 – CMake build options for Embree.

Option	Description	Default
BUILD_EMBREE_SHARED_LIB	Build Embree as a shared library.	ON
BUILD_TUTORIALS	Builds the C++ version of the Embree tutorials.	ON
BUILD_TUTORIALS_ISPC	Builds the ISPC version of the Embree tutorials.	ON
CMAKE_BUILD_TYPE	Can be used to switch between Debug mode (Debug) and Release mode (Release).	Release
COMPILER	Select either GCC, ICC, or CLANG as compiler.	GCC
RTCORE_BACKFACE_CULLING	Enables backface culling, i.e. only surfaces facing a ray can be hit.	OFF
RTCORE_BUFFER_STRIDE	Enables the buffer stride feature.	ON
RTCORE_INTERSECTION_FILTER	Enables the intersection filter feature.	ON
RTCORE_RAY_MASK	Enables the ray masking feature.	OFF
RTCORE_SPINLOCKS	Enables faster spinlocks for some builders.	OFF
XEON_ISA	Select highest supported ISA on Xeon™ CPUs (SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX-I, or AVX2).	AVX2
XEON_PHI_ISA	Enables generation of Xeon Phi™ version of kernels and tutorials (when BUILD_TUTORIALS is ON).	OFF

You need at least Intel Compiler 11.1 or GCC 4.7.

2.2 Xeon Phi™

Embree supports the Xeon Phi™ coprocessor under Linux. To compile Embree for Xeon Phi you need to enable the `XEON_PHI_ISA` option in CMake and have the Intel Compiler and the Intel Manycore Platform Software Stack (MPSS) installed.

Enabling the buffer stride feature reduces performance for building spatial hierarchies on Xeon Phi.

2.3 Windows

Embree requires the Intel SPMD Program Compiler (ISPC) to compile. We have tested ISPC version 1.7.0 and 1.8.0, but more recent versions of ISPC should also work. You can download and install the ISPC binaries from ispc.github.io. After installation, put the path to `ispc.exe` permanently into your `PATH` environment variable or you need to correctly set the `ISPC_EXECUTABLE` variable during CMake configuration.

You additionally have to install CMake (version 2.8.12 or higher). Note that you need a native Windows CMake installation, because CMake under Cygwin cannot generate solution files for Visual Studio.

2.3.1 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. Most configuration parameters described for the Linux build can be set under Windows as well. Finally, click Generate to create the Visual Studio solution files.

For compilation of Embree under Windows use the generated Visual Studio solution file `embree.sln`. The solution is by default setup to use the Microsoft Compiler. You can switch to the Intel Compiler by right clicking onto the solution in the Solution Explorer and then selecting the Intel Compiler. We recommend using 64 bit mode and the Intel Compiler for best performance.

To build all projects of the solution it is recommend to build the CMake utility project `ALL_BUILD`, which depends on all projects. Using "Build Solution" would also build all other CMake utility projects (such as `INSTALL`), which is usually not wanted.

We recommend enabling syntax highlighting for the `.ispc` source and `.isph` header files. To do so open Visual Studio 2008, go to Tools ⇒ Options ⇒ Text Editor ⇒ File Extension and add the isph and ispc extension for the "Microsoft Visual C++" editor.

2.3.2 Using the Command Line

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

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

Chapter 3
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. Filter callback functions are supported, that get invoked for every intersection encountered during traversal.

The Embree API exists in a C++ and ISPC version. This document describes the C++ version of the API, the ISPC version is almost identical. The only differences are that the ISPC version needs some ISPC specific uniform type modifiers, and 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 sequenced 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 in release mode, but do so if Embree is compiled in debug mode.

Possible error codes returned by `rtcGetError` are:

Table 3.1 – Return values of `rtcGetError`.

Error Code	Description
RTC_NO_ERROR	No error occurred.
RTC_UNKNOWN_ERROR	An unknown error has occurred.
RTC_INVALID_ARGUMENT	An invalid argument was specified.
RTC_INVALID_OPERATION	The operation is not allowed for the specified object.
RTC_OUT_OF_MEMORY	There is not enough memory left to complete the operation.
RTC_UNSUPPORTED_CPU	The CPU is not supported as it does not support SSE2.

Using the `rtcSetErrorFunction` call, it is also possible to set a callback function that is called whenever an error occurs. The callback function gets passed the error code, as well as some string that describes the error further. Passing `NULL` to `rtcSetErrorFunction` disables the set callback function again. The previously described error flags are also set if an error callback function is present.

3.1 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.

Table 3.2 – Dynamic type flags for `rtcNewScene`.

Scene Flag	Description
RTC_SCENE_STATIC	Scene is optimized for static geometry.
RTC_SCENE_DYNAMIC	Scene is 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, `rtcUpdate` or `rtcUpdateBuffer` has to get called for that geometry to specify which buffers got modified. Using multiple invocations of `rtcUpdateBuffer` the modified buffers can specified directly, while the `rtcUpdate` function simply tags each buffer of some geometry as modified. If geometries got enabled, disabled, deleted, or modified an `rtcCommit` call has to get invoked before performing any ray queries for the scene, otherwise the effect of the ray query is undefined.

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.

Table 3.3 – Acceleration structure flags for `rtcNewScene`.

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.

Table 3.4 – Traversal algorithm flags for `rtcNewScene`.

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.

Table 3.5 – Enabled algorithm flags for `rtcNewScene`.

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.

Algorithm Flag	Description

3.2 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`), hair geometries (`rtcNewHairGeometry`), 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.

3.2.1 Triangle Meshes

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

The number of triangles, the number of vertices, and optionally the 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 how to create a triangle mesh without motion blur:

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

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

Table 3.6 – Flags for the creation of new geometries.

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 writ-

ing into the vertex buffer (RTC_VERTEX_BUFFER). The index buffer contains an array of three 32 bit indices, while the vertex buffer contains an array of three float values 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);
```

Also see tutorial00 for an example of how to create triangle meshes.

3.2.2 Subdivision Surfaces

Catmull Clark subdivision surfaces for meshes consisting of triangle and quad primitives (even mixed inside one mesh) are supported, including support for edge creases, vertex creases, holes, and non-manifold geometry.

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

```
unsigned rtcNewSubdivisionMesh(RTCScene scene,
                               RTCGeometryFlags flags,
                               size_t numFaces,
                               size_t numEdges,
                               size_t numVertices,
                               size_t numEdgeCreases,
                               size_t numVertexCreases,
                               size_t numCorners,
                               size_t numHoles,
                               size_t numTimeSteps);
```

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

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

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

Optionally, the application can fill the level buffer (RTC_LEVEL_BUFFER) with a tessellation level for each of the numEdges edges. The subdivision level is a positive floating point value, that specifies how many quads along the edge should get generated during tessellation. The tessellation level is a lower bound, thus the implementation is free to choose a larger level. If no level buffer is specified a level of 1 is used.

Optionally, the application can fill the sparse edge crease buffers to make some edges appear sharper. The edge crease index buffer (RTC_EDGE_CREASE_

INDEX_BUFFER) contains numEdgeCreases many pairs of 32 bit vertex indices that specify unoriented edges. The edge crease weight buffer (RTC_EDGE_CREASE_WEIGHT_BUFFER) stores for each of theses crease edges a positive floating point weight. The larger this weight, the sharper the edge. Specifying a weight of infinity is supported and marks an edge as infinitely sharp. Storing an edge multiple times with the same crease weight is allowed, but has lower performance. Storing the an edge multiple times with different crease weights results in undefined behavior. For a stored edge (i,j), the reverse direction edges (j,i) does not have to get stored, as both are considered the same edge.

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

Also see tutorial08 for an example of how to create subdivision surfaces.

3.2.3 Hair Geometry

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

Hair geometries are created using the `rtcNewHairGeometry` function call, and potentially deleted using the `rtcDeleteGeometry` function call.

The number of hair curves, the number of vertices, and optionally the number of time steps (1 for normal curves, and 2 for linear motion blur) have to get specified at construction time.

The curve indices can be set by mapping and writing to the index buffer (RTC_INDEX_BUFFER) and the control vertices can be set by mapping and writing into the vertex buffer (RTC_VERTEX_BUFFER). In case of linear motion blur, two vertex buffers (RTC_VERTEX_BUFFER0 and RTC_VERTEX_BUFFER1) have to get filled, one for each time step.

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

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

The following example demonstrates how to create some hair geometry:

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

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

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

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

Also see tutorial07 for an example of how to create and use hair geometry.

3.2.4 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 invocation 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 following example illustrates creating an array with two user geometries:

```
struct UserObject { ... };

void userBoundsFunction(UserObject* userGeom, size_t i, RTCBounds& bounds) {
  bounds = <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 should 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.

3.2.5 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:

Table 3.7 – Matrix layouts for `rtcSetTransform`.

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

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

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

See tutorial04 for an example of how to use instances.

3.3 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:

Table 3.8 – Data fields of a ray.

Member	In/Out	Description
org	in	ray origin
dir	in	ray direction (can be unnormalized)
tnear	in	start of ray segment
tfar	in/out	end of ray segment, set to hit distance after intersection
time	in	time used for motion blur
mask	in	ray mask to mask out geometries
Ng	out	unnormalized 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 $[0, \infty)$, 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.f;
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.f;
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. Other hit information of the ray is undefined after calling `rtcOccluded`.

See tutorial00 for an example of how to trace rays.

3.4 Buffer Sharing

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

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

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

The `offset` parameter specifies a byte offset to the start of the first element and the `stride` parameter specifies a byte stride between the different elements of the shared buffer. This support for offset and stride allows the application quite some freedom in the data layout of these buffers, however, some restrictions apply. Index buffers always store 32 bit indices and vertex buffers always store single precision floating point data. The start address ptr+offset and stride always have to be aligned to 4 bytes on Xeon CPUs and 16 bytes on Xeon Phi™ accelerators, otherwise the `rtcSetBuffer` function will fail. For vertex buffers, the 4 bytes after the z-coordinate of the last vertex have to be readable memory, thus padding is required for some layouts.

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

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

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

The support for offset and stride is enabled by default, but can get disabled at compile time using the `RTCORE_BUFFER_STRIDE` parameter in CMake. Disabling this feature enables the default offset and stride which increases performance of spatial index structure build, thus can be useful for dynamic content.

3.5 Linear Motion Blur

A triangle mesh or hair geometry with linear motion blur support is created by setting the number of time steps to 2 at geometry construction time. Specifying a number of time steps of 0 or larger than 2 is invalid. For a triangle mesh or hair geometry with linear motion blur, the user has to set the `RTC_VERTEX_BUFFER0` and `RTC_VERTEX_BUFFER1` vertex arrays, one for each time step.

```
unsigned geomID = rtcNewTriangleMesh(scene, geomFlags, numTris, numVertices, 2);
rtcSetBuffer(scene, geomID, RTC_VERTEX_BUFFER0, vertex0Ptr, 0, sizeof(Vertex));
rtcSetBuffer(scene, geomID, RTC_VERTEX_BUFFER1, vertex1Ptr, 0, sizeof(Vertex));
rtcSetBuffer(scene, geomID, RTC_INDEX_BUFFER, indexPtr, 0, sizeof(Triangle));
```

If a scene contains geometries 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.

3.6 Geometry Mask

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

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

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

3.7 Filter Functions

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

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

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

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

3.8 Displacement Mapping Functions

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

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

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

The displacement function has to have the following type:

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

The displacement function is called with the user data pointer of the geometry (ptr), the geometry ID (geomID) and primitive ID (primID) of a patch to displace. For this patch, a number N of points to displace are specified in a struct of array layout. For each point to displace the local patch UV coordinates (u and v arrays), the geometry normal (nx, ny, and nz arrays), as well as world space position (px, py, and pz arrays) are provided. The task of the displacement function is to use this information and move the world space position inside the allowed specified bounds around the point.

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

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

3.9 Sharing Threads with Embree

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

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

API call to commit the scene. This function has to get called by all threads that want to cooperate in the scene commit. Each call is provided the scene to commit, the index of the calling thread in the range [0, `threadCount`-1], and the number of threads that will call into this commit operation for the scene. Multiple such scene commit operations can also be running at the same time, e.g. it is possible to commit many small scenes in parallel using one thread per commit operation. Subsequent commit operations for the same scene can use different number of threads or the Embree internal threads using the

```
void rtcCommitThread()
```

call.

Chapter 4
Embree Tutorials

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

Under Linux Embree also comes with an ISPC version of all tutorials for the Intel® Xeon Phi™ coprocessor. The executables of this version of the tutorials are named `tutorialXX_xeonphi` and only work if a Xeon Phi coprocessor is present in the system. The Xeon Phi version of the tutorials get started on the host CPU, just like all other tutorials, and will connect automatically to one installed Xeon Phi coprocessor in the system.

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

```
./tutorial00 -vp 10 10 10 -vi 0 0 0
```

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

```
./tutorial00 -size 1024 1024
./tutorial00 -fullscreen
```

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

```
./tutorial00 -rtcore verbose=2,threads=1,accel=bvh4.triangle1
```

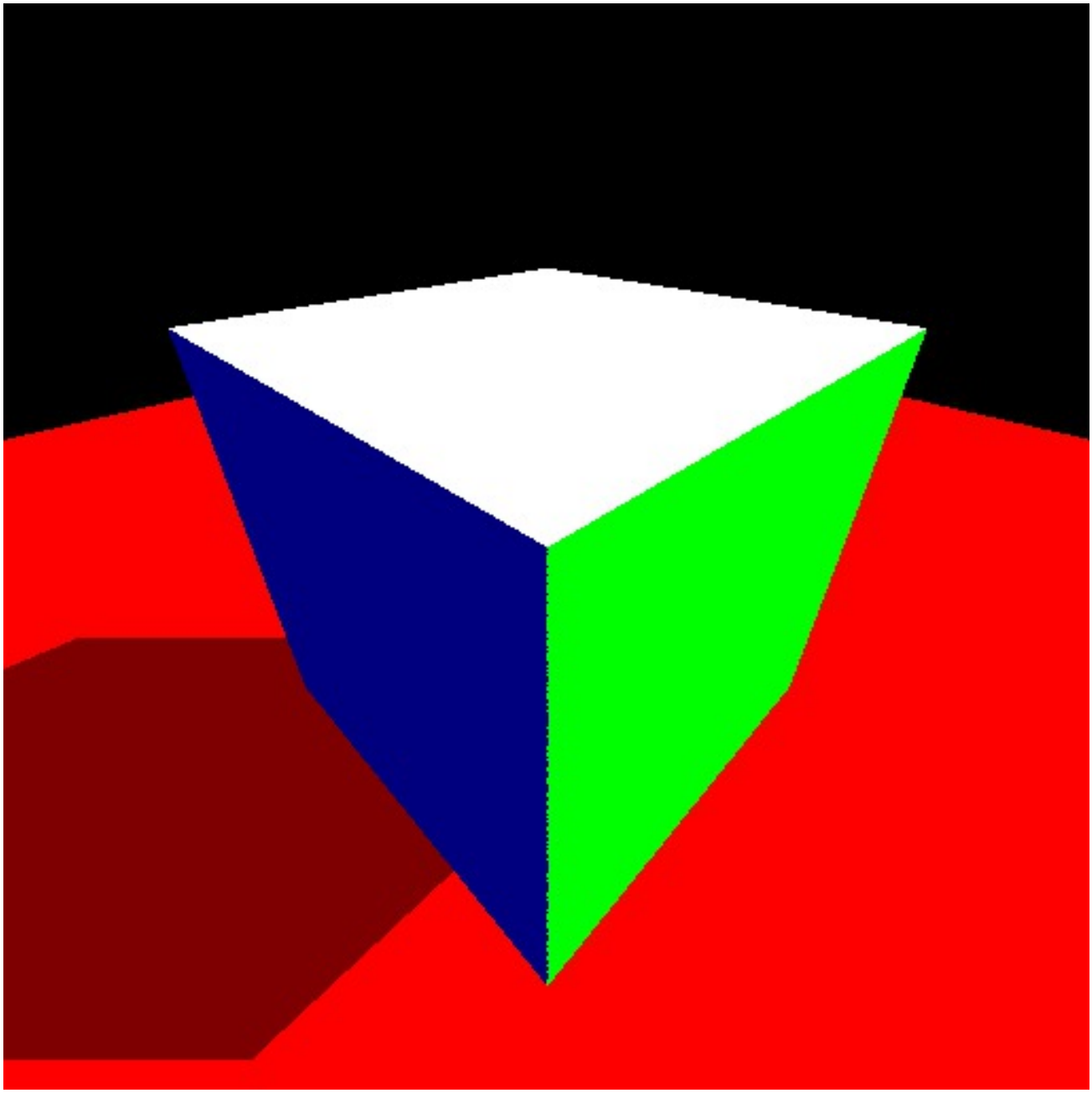
The navigation in the interactive display mode follows the camera orbit model, where the camera revolves around the current center of interest. With the left mouse button you can rotate around the center of interest (the point initially set with `-vi`). Holding Control pressed while clicking the left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

You can use the following keys:

F1 Default shading
F2 Gray EyeLight shading
F3 Wireframe shading
F4 UV Coordinate visualization
F5 Geometry normal visualization
F6 Geometry ID visualization

F7 Geometry ID and Primitive ID visualization
F8 Simple shading with 16 rays per pixel for benchmarking.
F9 Switches to render cost visualization. Pressing again reduces brightness.
F10 Switches to render cost visualization. Pressing again increases brightness.
f Enters or leaves full screen mode.
c Prints camera parameters.
ESC Exists the tutorial.
q Exists the tutorial.

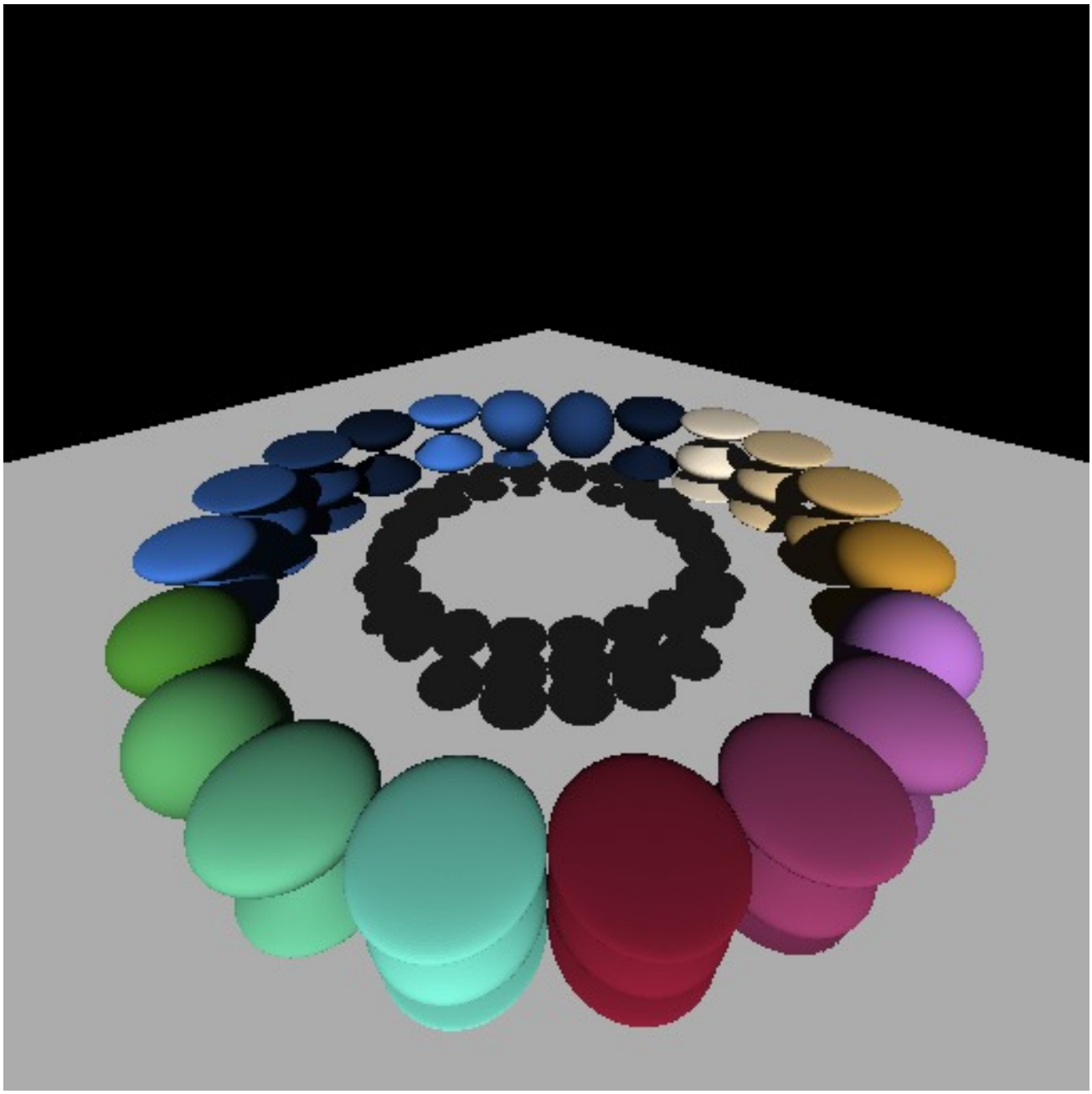
4.1 Tutorial00



This tutorial demonstrates the creation of a static cube and ground plane

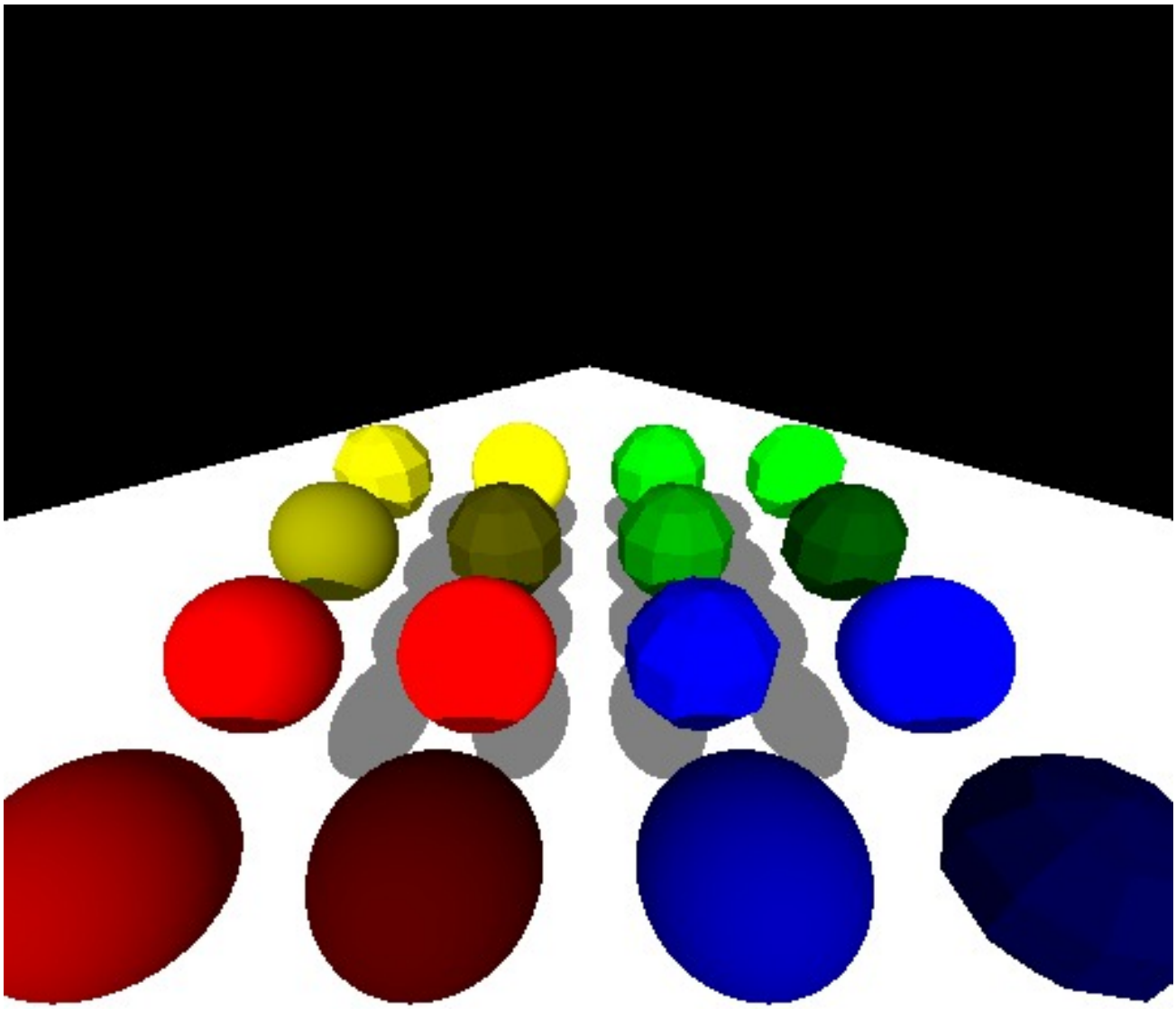
using triangle meshes. It also demonstrates the use of the `rtcIntersect` and `rtcOccluded` functions to render primary visibility and hard shadows. The cube sides are colored based on the ID of the hit primitive.

4.2 Tutorial01



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

4.3 Tutorial02



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

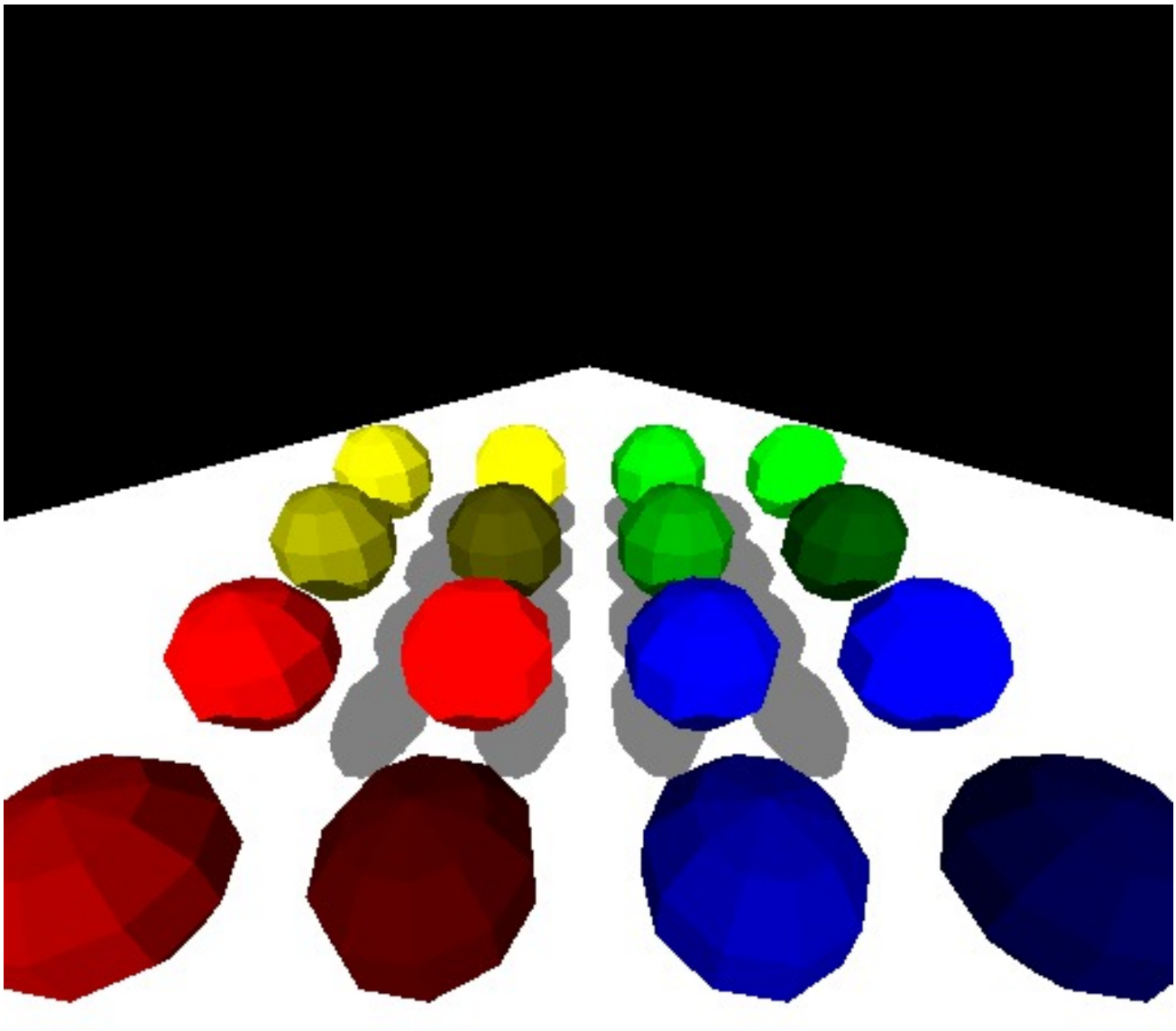
4.4 Tutorial03

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

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

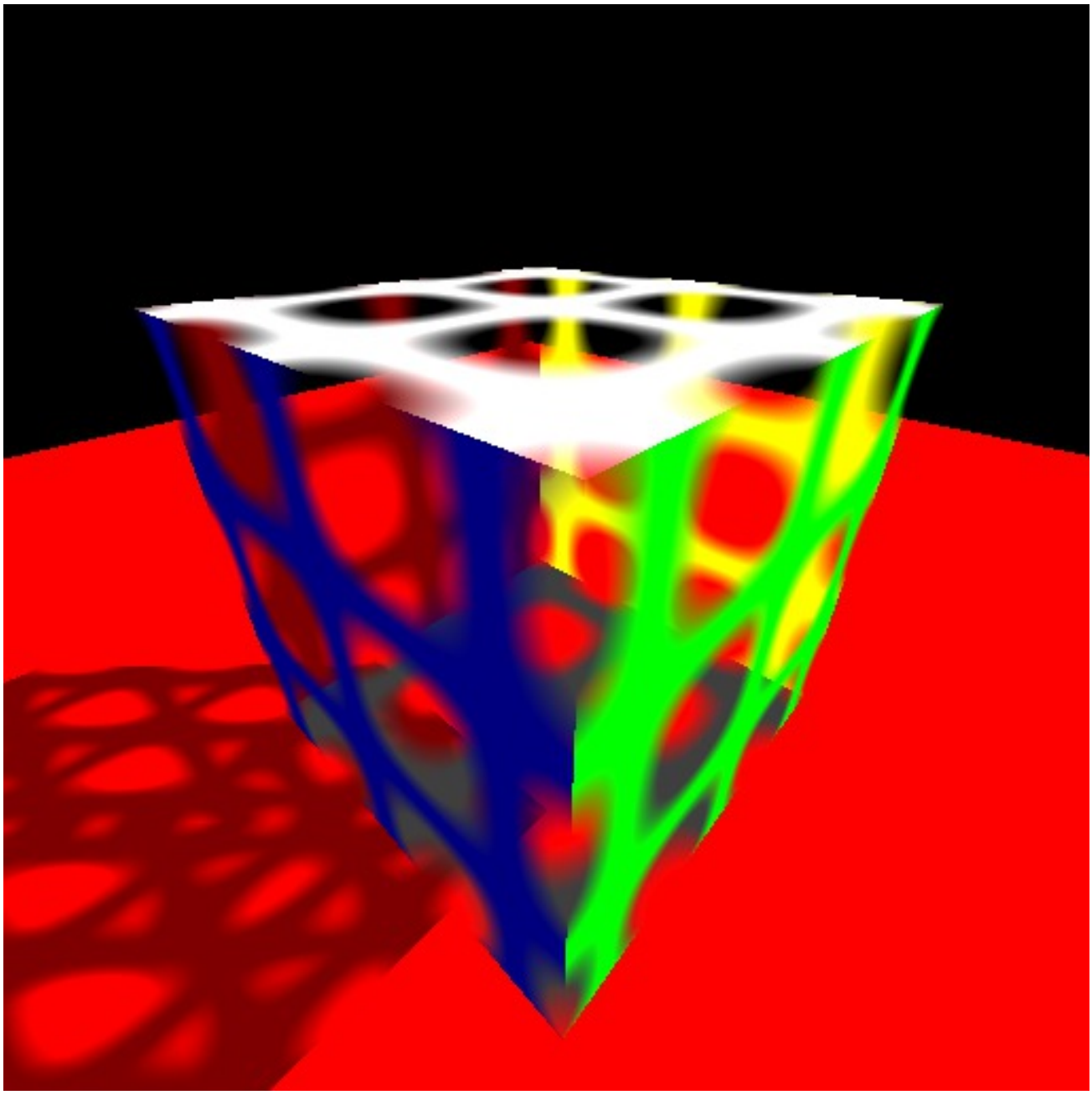
```
./tutorial03 -i model.obj
```

4.5 Tutorial04



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

4.6 Tutorial05



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.

4.7 Tutorial06

This tutorial is a simple path tracer, building on tutorial03.

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

```
./tutorial06 -i model.obj -ambientlight 1 1 1
```

4.8 Tutorial07

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

4.9 Tutorial08

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