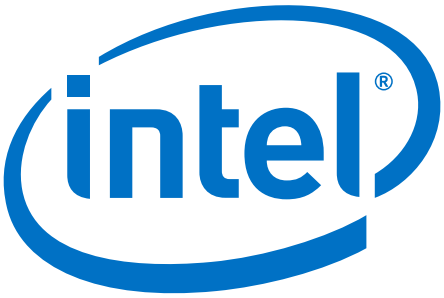

# INTEL® OSPRAY

## AN OPEN, SCALABLE, PARALLEL, RAY TRACING BASED RENDERING ENGINE FOR HIGH-FIDELITY VISUALIZATION

Version 2.3.0
September 14, 2020

# Contents

# Chapter 1
# OSPRay Overview

Intel OSPRay is an **o**pen source, **s**calable, and **p**ortable **ray** tracing engine for high-performance, high-fidelity visualization on Intel Architecture CPUs. OSPRay is part of the Intel oneAPI Rendering Toolkit and is released under the permissive Apache 2.0 license.

The purpose of OSPRay is to provide an open, powerful, and easy-to-use rendering library that allows one to easily build applications that use ray tracing based rendering for interactive applications (including both surface- and volume-based visualizations). OSPRay is completely CPU-based, and runs on anything from laptops, to workstations, to compute nodes in HPC systems.

OSPRay internally builds on top of Intel Embree and ISPC (Intel SPMD Program Compiler), and fully exploits modern instruction sets like Intel SSE4, AVX, AVX2, and AVX-512 to achieve high rendering performance, thus a CPU with support for at least SSE4.1 is required to run OSPRay.

## 1.1 OSPRay Support and Contact

OSPRay is under active development, and though we do our best to guarantee stable release versions a certain number of bugs, as-yet-missing features, inconsistencies, or any other issues are still possible. Should you find any such issues please report them immediately via OSPRay's GitHub Issue Tracker (or, if you should happen to have a fix for it,you can also send us a pull request); for missing features please contact us via email at `ospray@googlegroups.com`.

To receive release announcements simply "Watch" the OSPRay repository on GitHub.

### Changes in v2.3.0:

- Re-add SciVis renderer features (the previous version is still available as `ao` renderer)
  - Lights are regarded, and thus the OBJ material terms `ks` and `ns` have effect again
  - Hard shadows are enabled via the `shadows` parameter
  - The control of ambient occlusion changed:
    * The `aoIntensity` parameter is replaced by the combined intensity of ambient lights in the `World`
    * The effect range is controlled via `aoDistance`
- Added support for data arrays with a stride between voxels in volumes
- Application thread waiting for finished image via `ospWait` participates in rendering, increasing CPU utilization; via rkcommon v1.5.0

- Added `ospray_cpp` compatibility headers for C++ wrappers to understand rkcommon and glm short vector types
  - For rkcommon, include `ospray/ospray_cpp/ext/rkcommon.h`
  - For glm, include `ospray/ospray_cpp/ext/glm.h`
  - Note in debug builds some compilers will not optimize out type trait definitions. This will require users to manually instantiate the glm definitions in one translation unit within the application using `#define OSPRAY_GLM_DEFINITIONS` before including `ext/glm.h`: see `ospTutorialGLM` as an example
- Changed parameters to `volume` texture: it now directly accepts the `volume` and the `transferFunction`
- Fixed many memory leaks
- Handle `NaN` during volume sampling, which led to bounding boxes being visible for some volumes and settings
- Depth is now "accumulated" as well, using the minimum
- Fix shading for multiple modes of the `debug` renderer
- New minimum ISPC version is 1.14.1

## Changes in v2.2.0:

- Support for texture transformation in SciVis OBJ material
- Add transformations for volume textures; volume texture lookups are now with local object coordinates (not world coordinates anymore)
- Changed behavior: if solely a texture is given, then the default value of the corresponding parameter is *not* multiplied
- Support for better antialiasing using a set of different pixel filters (e.g, box, Gaussian, ...). The size of the pixel filter is defined by the used filter type. Previously OSPRay implicitly used a box filter with a size of 1, for better results the default filter is now `OSP_PIXELFILTER_GAUSS`
- Support stereo3d mode for panoramic camera
- Add new camera `stereoMode` `OSP_STEREO_TOP_BOTTOM` (with left eye at top half of the image)
- Added support for random light sampling to the `pathtracer`, the number of sampled light sources per path vertex is defined by the `lightSamples` parameter
- Support ring light by extending spot with `innerRadius`
- Fixed nonphysical behavior of the `spot` and `sphere` light sources
  - for area lights (when `radius > 0`) surfaces close to the light will be darker
  - the `spot` now has an angular falloff, such that a disk light is a proper lambertian area light, which leads to darker regions perpendicular to its direction (thus barely visible with a typically small `openingAngle`)
- Support for Open VKL v0.10.0 and its new sampler object API, thus this is now the required minimum version
- Added support for particle and VDB volumes
- Move from `ospcommon` to `rkcommon` v1.4.2
- New minimum ISPC version is 1.10.0
- Status and error callbacks now support a user pointer
- Enabled C++ wrappers (`ospray_cpp`) to work with non-rkcommon math types
  - Note that while the C API remains the same, the C++ wrappers will require some application updates to account for these changes
- Fix bug where `ospGetCurrentDevice` would crash if used before `ospInit`

- Allow `NULL` handles to be passed to `ospDeviceRetain` and `ospDeviceRelease`
- ISPC generated headers containing the exported functions for OSPRay's ISPC types and functions are now distributed with the SDK
- Add CarPaint `flakeColor` parameter, defaults to current Aluminium
- Fixed Debug build (which were producing different images)
- The path tracer now also regards the renderer materialist when creating geometry lights

## Changes in v2.1.1:

- CarPaint material obeys `coat` weight parameter
- Correct depth buffer values with SciVis renderer
- Adaptions to Embree v3.10.0
- The Linux binary release finds `ospcommon` again

For the complete history of changes have a look at the CHANGELOG.

# Chapter 2
# Building and Finding OSPRay

The latest OSPRay sources are always available at the OSPRay GitHub repository. The default `master` branch should always point to the latest bugfix release.

## 2.1 Prerequisites

OSPRay currently supports Linux, Mac OS X, and Windows. In addition, before you can build OSPRay you need the following prerequisites:

- You can clone the latest OSPRay sources via:

  ```
  git clone https://github.com/ospray/ospray.git
  ```

- To build OSPRay you need CMake, any form of C++11 compiler (we recommend using GCC, but also support Clang, MSVC, and Intel® C++ Compiler (icc)), and standard Linux development tools. To build the interactive tutorials, you should also have some version of OpenGL and GLFW.

- Additionally you require a copy of the Intel® SPMD Program Compiler (ISPC), version 1.14.1 or later. Please obtain a release of ISPC from the ISPC downloads page. The build system looks for ISPC in the `PATH` and in the directory right "next to" the checked-out OSPRay sources.[1] Alternatively set the CMake variable `ISPC_EXECUTABLE` to the location of the ISPC compiler.

- OSPRay builds on top of the Intel oneAPI Rendering Toolkit common library `rkcommon`. The library provides abstractions for tasking, aligned memory allocation, vector math types, among others. For users who also need to build rkcommon, we recommend the default the Intel Threading Building Blocks (TBB) as tasking system for performance and flexibility reasons. Alternatively you can set CMake variable `RKCOMMON_TASKING_SYSTEM` to `OpenMP` or `Internal`.

- OSPRay also heavily uses Intel Embree, installing version 3.8.0 or newer is required. If Embree is not found by CMake its location can be hinted with the variable `embree_DIR`.

- OSPRay also heavily uses Intel Open VKL, installing version 0.10.0 or newer is required. If Open VKL is not found by CMake its location can be hinted with the variable `openvkl_DIR`.

- OSPRay also provides an optional module that adds support for Intel Open Image Denoise, which is enabled by `OSPRAY_MODULE_DENOISER`. When loaded, this module enables the `denosier` image operation. You may need

[1] For example, if OSPRay is in `~/Projects/ospray`, ISPC will also be searched in `~/Projects/ispc-v1.14.1-linux`

to hint the location of the library with the CMake variable `OpenImageDenoise_DIR`.

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

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

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

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

```
sudo apt-get install cmake-curses-gui
sudo apt-get install libtbb-dev
```

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

```
sudo port install cmake tbb
```

Under Windows please directly use the appropriate installers for CMake, TBB, ISPC (for your Visual Studio version) and Embree.

## 2.2 CMake Superbuild

For convenience, OSPRay provides a CMake Superbuild script which will pull down OSPRay's dependencies and build OSPRay itself. By default, the result is an install directory, with each dependency in its own directory.

Run with:

```
mkdir build
cd build
cmake [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
cmake --build .
```

On Windows make sure to select the non-default 64bit generator, e.g.

```
cmake -G "Visual Studio 15 2017 Win64" [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
```

The resulting `install` directory (or the one set with `CMAKE_INSTALL_PREFIX`) will have everything in it, with one subdirectory per dependency.

CMake options to note (all have sensible defaults):

**CMAKE_INSTALL_PREFIX** will be the root directory where everything gets installed.

**BUILD_JOBS** sets the number given to `make -j` for parallel builds.

**INSTALL_IN_SEPARATE_DIRECTORIES** toggles installation of all libraries in separate or the same directory.

**BUILD_EMBREE_FROM_SOURCE** set to OFF will download a pre-built version of Embree.

**BUILD_OIDN_FROM_SOURCE** set to OFF will download a pre-built version of Open Image Denoise.

**BUILD_OIDN_VERSION** determines which version of Open Image Denoise to pull down.

For the full set of options, run:

```
ccmake [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
```

or

```
cmake-gui [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
```

## 2.3 Standard CMake build

### 2.3.1 Compiling OSPRay on Linux and Mac OS X

Assuming the above requisites are all fulfilled, building OSPRay through CMake is easy:

- Create a build directory, and go into it

  ```
  mkdir ospray/build
  cd ospray/build
  ```

  (We do recommend having separate build directories for different configurations such as release, debug, etc.).

- The compiler CMake will use will default to whatever the `CC` and `CXX` environment variables point to. Should you want to specify a different compiler, run cmake manually while specifying the desired compiler. The default compiler on most linux machines is `gcc`, but it can be pointed to `clang` instead by executing the following:

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

  CMake will now use Clang instead of GCC. If you are ok with using the default compiler on your system, then simply skip this step. Note that the compiler variables cannot be changed after the first `cmake` or `ccmake` run.

- Open the CMake configuration dialog

  ```
  ccmake ..
  ```

- Make sure to properly set build mode and enable the components you need, etc.; then type 'c'onfigure and 'g'enerate. When back on the command prompt, build it using

  ```
  make
  ```

- You should now have `libospray.[so,dylib]` as well as a set of example applications.

### 2.3.2 Compiling OSPRay on Windows

On Windows using the CMake GUI (`cmake-gui.exe`) is the most convenient way to configure OSPRay and to create the Visual Studio solution files:

- Browse to the OSPRay sources and specify a build directory (if it does not exist yet CMake will create it).

- Click "Configure" and select as generator the Visual Studio version you have (OSPRay needs Visual Studio 14 2015 or newer), for Win64 (32 bit builds are not supported by OSPRay), e.g., "Visual Studio 15 2017 Win64".

- If the configuration fails because some dependencies could not be found then follow the instructions given in the error message, e.g., set the variable `embree_DIR` to the folder where Embree was installed and `openvkl_DIR` to where Open VKL was installed.

- Optionally change the default build options, and then click "Generate" to create the solution and project files in the build directory.

- Open the generated `OSPRay.sln` in Visual Studio, select the build configuration and compile the project.

Alternatively, OSPRay can also be built without any GUI, entirely on the console. In the Visual Studio command prompt type:

```
cd path\to\ospray
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" [-D VARIABLE=value] ..
cmake --build . --config Release
```

Use `-D` to set variables for CMake, e.g., the path to Embree with "`-D embree_DIR=\path\to\embree`".

You can also build only some projects with the `--target` switch. Additional parameters after "`--`" will be passed to `msbuild`. For example, to build in parallel only the OSPRay library without the example applications use

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

## 2.4 Finding an OSPRay Install with CMake

Client applications using OSPRay can find it with CMake's `find_package()` command. For example,

```
find_package(ospray 2.0.0 REQUIRED)
```

finds OSPRay via OSPRay's configuration file `osprayConfig.cmake`[2]. Once found, the following is all that is required to use OSPRay:

```
target_link_libraries(${client_target} ospray::ospray)
```

This will automatically propagate all required include paths, linked libraries, and compiler definitions to the client CMake target (either an executable or library).

Advanced users may want to link to additional targets which are exported in OSPRay's CMake config, which includes all installed modules. All targets built with OSPRay are exported in the `ospray::` namespace, therefore all targets locally used in the OSPRay source tree can be accessed from an install. For example, `ospray_module_ispc` can be consumed directly via the `ospray::ospray_module_ispc` target. All targets have their libraries, includes, and definitions attached to them for public consumption (please report bugs if this is broken!).

[2] This file is usually in `${install_location}/[lib|lib64]/cmake/ospray-${version}/`. If CMake does not find it automatically, then specify its location in variable `ospray_DIR` (either an environment variable or CMake variable).

# Chapter 3
# OSPRay API

To access the OSPRay API you first need to include the OSPRay header

```
#include "ospray/ospray.h"
```

where the API is compatible with C99 and C++.

## 3.1 Initialization and Shutdown

To use the API, OSPRay must be initialized with a "device". A device is the object which implements the API. Creating and initializing a device can be done in either of two ways: command line arguments using `ospInit` or manually instantiating a device and setting parameters on it.

### 3.1.1 Command Line Arguments

The first is to do so by giving OSPRay the command line from `main()` by calling

```
OSPError ospInit(int *argc, const char **argv);
```

OSPRay parses (and removes) its known command line parameters from your application's `main` function. For an example see the tutorial. For possible error codes see section Error Handling and Status Messages. It is important to note that the arguments passed to `ospInit()` are processed in order they are listed. The following parameters (which are prefixed by convention with "`--osp:`") are understood:

### 3.1.2 Manual Device Instantiation

The second method of initialization is to explicitly create the device and possibly set parameters. This method looks almost identical to how other objects are created and used by OSPRay (described in later sections). The first step is to create the device with

```
OSPDevice ospNewDevice(const char *type);
```

where the `type` string maps to a specific device implementation. OSPRay always provides the "`cpu`" device, which maps to a fast, local CPU implementation. Other devices can also be added through additional modules, such as distributed MPI device implementations.

Once a device is created, you can call

```
void ospDeviceSetParam(OSPObject, const char *id, OSPDataType type, const void *mem);
```

Table 3.1 – Command line parameters accepted by OSPRay's `ospInit`.

| Parameter | Description |
|---|---|
| `--osp:debug` | enables various extra checks and debug output, and disables multi-threading |
| `--osp:num-threads=<n>` | use `n` threads instead of per default using all detected hardware threads |
| `--osp:log-level=<str>` | set logging level; valid values (in order of severity) are `none`, `error`, `warning`, `info`, and `debug` |
| `--osp:warn-as-error` | send `warning` and `error` messages through the error callback, otherwise send `warning` messages through the message callback; must have sufficient `logLevel` to enable warnings |
| `--osp:verbose` | shortcut for `--osp:log-level=info` and enable debug output on `cout`, error output on `cerr` |
| `--osp:vv` | shortcut for `--osp:log-level=debug` and enable debug output on `cout`, error output on `cerr` |
| `--osp:load-modules=<name>[,...]` | load one or more modules during initialization; equivalent to calling `ospLoadModule(name)` |
| `--osp:log-output=<dst>` | convenience for setting where status messages go; valid values for `dst` are `cerr` and `cout` |
| `--osp:error-output=<dst>` | convenience for setting where error messages go; valid values for `dst` are `cerr` and `cout` |
| `--osp:device=<name>` | use `name` as the type of device for OSPRay to create; e.g., `--osp:device=cpu` gives you the default `cpu` device; Note if the device to be used is defined in a module, remember to pass `--osp:load-modules=<name>` first |
| `--osp:set-affinity=<n>` | if `1`, bind software threads to hardware threads; `0` disables binding; default is `1` on KNL and `0` otherwise |
| `--osp:device-params=<param>:<value>[,...]` | set one or more other device parameters; equivalent to calling `ospDeviceSet*(param, value)` |

Table 3.2 – Parameters shared by all devices.

| Type | Name | Description |
|---|---|---|
| int | numThreads | number of threads which OSPRay should use |
| int | logLevel | logging level; valid values (in order of severity) are `OSP_LOG_NONE`, `OSP_LOG_ERROR`, `OSP_LOG_WARNING`, `OSP_LOG_INFO`, and `OSP_LOG_DEBUG` |
| string | logOutput | convenience for setting where status messages go; valid values are `cerr` and `cout` |
| string | errorOutput | convenience for setting where error messages go; valid values are `cerr` and `cout` |
| bool | debug | set debug mode; equivalent to `logLevel=debug` and `numThreads=1` |
| bool | warnAsError | send `warning` and `error` messages through the error callback, otherwise send `warning` messages through the message callback; must have sufficient `logLevel` to enable warnings |
| bool | setAffinity | bind software threads to hardware threads if set to 1; 0 disables binding omitting the parameter will let OSPRay choose |

to set parameters on the device. The semantics of setting parameters is exactly the same as `ospSetParam`, which is documented below in the parameters section. The following parameters can be set on all devices:

Once parameters are set on the created device, the device must be committed

with

```
void ospDeviceCommit(OSPDevice);
```

To use the newly committed device, you must call

```
void ospSetCurrentDevice(OSPDevice);
```

This then sets the given device as the object which will respond to all other OSPRay API calls.

Device handle lifetimes are managed with two calls, the first which increments the internal reference count to the given `OSPDevice`

```
void ospDeviceRetain(OSPDevice)
```

and the second which decrements the reference count

```
void ospDeviceRelease(OSPDevice)
```

Users can change parameters on the device after initialization (from either method above), by calling

```
OSPDevice ospGetCurrentDevice();
```

This function returns the handle to the device currently used to respond to OSPRay API calls, where users can set/change parameters and recommit the device. If changes are made to the device that is already set as the current device, it does not need to be set as current again. Note this API call will increment the ref count of the returned device handle, so applications must use `ospDeviceRelease` when finished using the handle to avoid leaking the underlying device object. If there is no current device set, this will return an invalid NULL handle.

When a device is created, its reference count is initially 1. When a device is set as the current device, it internally has its reference count incremented. Note that `ospDeviceRetain` and `ospDeviceRelease` should only be used with reference counts that the application tracks: removing reference held by the current set device should be handled by `ospShutdown`. Thus, `ospDeviceRelease` should only decrement the reference counts that come from `ospNewDevice`, `ospGetCurrentDevice`, and the number of explicit calls to `ospDeviceRetain`.

OSPRay allows applications to query runtime properties of a device in order to do enhanced validation of what device was loaded at runtime. The following function can be used to get these device-specific properties (attributes about the device, not parameter values)

```
int64_t ospDeviceGetProperty(OSPDevice, OSPDeviceProperty);
```

It returns an integer value of the queried property and the following properties can be provided as parameter:

```
OSP_DEVICE_VERSION
OSP_DEVICE_VERSION_MAJOR
OSP_DEVICE_VERSION_MINOR
OSP_DEVICE_VERSION_PATCH
OSP_DEVICE_SO_VERSION
```

### 3.1.3 Environment Variables

OSPRay's generic device parameters can be overridden via environment variables for easy changes to OSPRay's behavior without needing to change the application (variables are prefixed by convention with "`OSPRAY_`"):

Note that these environment variables take precedence over values specified through `ospInit` or manually set device parameters.

Table 3.3 – Environment variables interpreted by OSPRay.

| Variable | Description |
|---|---|
| OSPRAY_NUM_THREADS | equivalent to `--osp:num-threads` |
| OSPRAY_LOG_LEVEL | equivalent to `--osp:log-level` |
| OSPRAY_LOG_OUTPUT | equivalent to `--osp:log-output` |
| OSPRAY_ERROR_OUTPUT | equivalent to `--osp:error-output` |
| OSPRAY_DEBUG | equivalent to `--osp:debug` |
| OSPRAY_WARN_AS_ERROR | equivalent to `--osp:warn-as-error` |
| OSPRAY_SET_AFFINITY | equivalent to `--osp:set-affinity` |
| OSPRAY_LOAD_MODULES | equivalent to `--osp:load-modules`, can be a comma separated list of modules which will be loaded in order |
| OSPRAY_DEVICE | equivalent to `--osp:device:` |

### 3.1.4 Error Handling and Status Messages

The following errors are currently used by OSPRay:

| Name | Description |
|---|---|
| OSP_NO_ERROR | no error occurred |
| OSP_UNKNOWN_ERROR | an unknown error occurred |
| OSP_INVALID_ARGUMENT | an invalid argument was specified |
| OSP_INVALID_OPERATION | the operation is not allowed for the specified object |
| OSP_OUT_OF_MEMORY | there is not enough memory to execute the command |
| OSP_UNSUPPORTED_CPU | the CPU is not supported (minimum ISA is SSE4.1) |
| OSP_VERSION_MISMATCH | a module could not be loaded due to mismatching version |

Table 3.4 – Possible error codes, i.e., valid named constants of type `OSPError`.

These error codes are either directly return by some API functions, or are recorded to be later queried by the application via

```
OSPError ospDeviceGetLastErrorCode(OSPDevice);
```

A more descriptive error message can be queried by calling

```
const char* ospDeviceGetLastErrorMsg(OSPDevice);
```

Alternatively, the application can also register a callback function of type

```
typedef void (*OSPErrorCallback)(void *userData, OSPError, const char* errorDetails);
```

via

```
void ospDeviceSetErrorCallback(OSPDevice, OSPErrorCallback, void *userData);
```

to get notified when errors occur.

Applications may be interested in messages which OSPRay emits, whether for debugging or logging events. Applications can call

```
void ospDeviceSetStatusCallback(OSPDevice, OSPStatusCallback, void *userData);
```

in order to register a callback function of type

```
typedef void (*OSPStatusCallback)(void *userData, const char* messageText);
```

which OSPRay will use to emit status messages. By default, OSPRay uses a callback which does nothing, so any output desired by an application will require that a callback is provided. Note that callbacks for C++ `std::cout` and `std::cerr` can be alternatively set through `ospInit()` or the `OSPRAY_LOG_OUTPUT` environment variable.

Applications can clear either callback by passing `nullptr` instead of an actual function pointer.

### 3.1.5 Loading OSPRay Extensions at Runtime

OSPRay's functionality can be extended via plugins (which we call "modules"), which are implemented in shared libraries. To load module `name` from `libospray_module_<name>.so` (on Linux and Mac OS X) or `ospray_module_<name>.dll` (on Windows) use

```
OSPError ospLoadModule(const char *name);
```

Modules are searched in OS-dependent paths. `ospLoadModule` returns `OSP_NO_ERROR` if the plugin could be successfully loaded.

### 3.1.6 Shutting Down OSPRay

When the application is finished using OSPRay (typically on application exit), the OSPRay API should be finalized with

```
void ospShutdown();
```

This API call ensures that the current device is cleaned up appropriately. Due to static object allocation having non-deterministic ordering, it is recommended that applications call `ospShutdown()` before the calling application process terminates.

## 3.2 Objects

All entities of OSPRay (the renderer, volumes, geometries, lights, cameras, ...) are a logical specialization of `OSPObject` and share common mechanism to deal with parameters and lifetime.

An important aspect of object parameters is that parameters do not get passed to objects immediately. Instead, parameters are not visible at all to objects until they get explicitly committed to a given object via a call to

```
void ospCommit(OSPObject);
```

at which time all previously additions or changes to parameters are visible at the same time. If a user wants to change the state of an existing object (e.g., to change the origin of an already existing camera) it is perfectly valid to do so, as long as the changed parameters are recommitted.

The commit semantic allow for batching up multiple small changes, and specifies exactly when changes to objects will occur. This can impact performance and consistency for devices crossing a PCI bus or across a network.

Note that OSPRay uses reference counting to manage the lifetime of all objects, so one cannot explicitly "delete" any object. Instead, to indicate that the application does not need and does not access the given object anymore, call

```
void ospRelease(OSPObject);
```

This decreases its reference count and if the count reaches 0 the object will automatically get deleted. Passing NULL is not an error.

Sometimes applications may want to have more than one reference to an object, where it is desirable for the application to increment the reference count of an object. This is done with

```
void ospRetain(OSPObject);
```

It is important to note that this is only necessary if the application wants to call ospRelease on an object more than once: objects which contain other objects as parameters internally increment/decrement ref counts and should not be explicitly done by the application.

### 3.2.1 Parameters

Parameters allow to configure the behavior of and to pass data to objects. However, objects do *not* have an explicit interface for reasons of high flexibility and a more stable compile-time API. Instead, parameters are passed separately to objects in an arbitrary order, and unknown parameters will simply be ignored (though a warning message will be posted). The following function allows adding various types of parameters with name id to a given object:

```
void ospSetParam(OSPObject, const char *id, OSPDataType type, const void *mem);
```

The valid parameter names for all OSPObjects and what types are valid are discussed in future sections.

Note that mem must always be a pointer *to* the object, otherwise accidental type casting can occur. This is especially true for pointer types (OSP_VOID_PTR and OSPObject handles), as they will implicitly cast to void *, but be incorrectly interpreted. To help with some of these issues, there also exist variants of ospSetParam for specific types, such as ospSetInt and ospSetVec3f in the OSPRay utility library (found in ospray_util.h).

Users can also remove parameters that have been explicitly set from ospSetParam. Any parameters which have been removed will go back to their default value during the next commit unless a new parameter was set after the parameter was removed. To remove a parameter, use

```
void ospRemoveParam(OSPObject, const char *id);
```

### 3.2.2 Data

OSPRay consumes data arrays from the application using a specific object type, OSPData. There are several components to describing a data array: element type, 1/2/3 dimensional striding, and whether the array is shared with the application or copied into opaque, OSPRay-owned memory.

Shared data arrays require that the application's array memory outlives the lifetime of the created OSPData, as OSPRay is referring to application memory. Where this is not preferable, applications use opaque arrays to allow the OSPData to own the lifetime of the array memory. However, opaque arrays dictate the cost of copying data into it, which should be kept in mind.

Thus, the most efficient way to specify a data array from the application is to created a shared data array, which is done with

```
OSPData ospNewSharedData(const void *sharedData,
    OSPDataType,
    uint64_t numItems1,
```

```
    int64_t byteStride1 = 0,
    uint64_t numItems2 = 1,
    int64_t byteStride2 = 0,
    uint64_t numItems3 = 1,
    int64_t byteStride3 = 0);
```

The call returns an `OSPData` handle to the created array. The calling program guarantees that the `sharedData` pointer will remain valid for the duration that this data array is being used. The number of elements `numItems` must be positive (there cannot be an empty data object). The data is arranged in three dimensions, with specializations to two or one dimension (if some `numItems` are 1). The distance between consecutive elements (per dimension) is given in bytes with `byteStride` and can also be negative. If `byteStride` is zero it will be determined automatically (e.g., as `sizeof(type)`). Strides do not need to be ordered, i.e., `byteStride2` can be smaller than `byteStride1`, which is equivalent to a transpose. However, if the stride should be calculated, then an ordering in dimensions is assumed to disambiguate, i.e., `byteStride1 < byteStride2 < byteStride3`.

The enum type `OSPDataType` describes the different element types that can be represented in OSPRay; valid constants are listed in the table below.

If the elements of the array are handles to objects, then their reference counter is incremented.

An opaque `OSPData` with memory allocated by OSPRay is created with

```
OSPData ospNewData(OSPDataType,
    uint32_t numItems1,
    uint32_t numItems2 = 1,
    uint32_t numItems3 = 1);
```

To allow for (partial) copies or updates of data arrays use

```
void ospCopyData(const OSPData source,
    OSPData destination,
    uint32_t destinationIndex1 = 0,
    uint32_t destinationIndex2 = 0,
    uint32_t destinationIndex3 = 0);
```

which will copy the whole[1] content of the `source` array into `destination` at the given location `destinationIndex`. The `OSPDataTypes` of the data objects must match. The region to be copied must be valid inside the destination, i.e., in all dimensions, `destinationIndex + sourceSize <= destinationSize`. The affected region `[destinationIndex, destinationIndex + sourceSize)` is marked as dirty, which may be used by OSPRay to only process or update that sub-region (e.g., updating an acceleration structure). If the destination array is shared with OSPData by the application (created with `ospNewSharedData`), then

- the source array must be shared as well (thus `ospCopyData` cannot be used to read opaque data)
- if source and destination memory overlaps (aliasing), then behavior is undefined
- except if source and destination regions are identical (including matching strides), which can be used by application to mark that region as dirty (instead of the whole `OSPData`)

To add a data array as parameter named `id` to another object call also use

```
void ospSetObject(OSPObject, const char *id, OSPData);
```

[1] The number of items to be copied is defined by the size of the source array

| Type/Name | Description |
|---|---|
| OSP_DEVICE | API device object reference |
| OSP_DATA | data reference |
| OSP_OBJECT | generic object reference |
| OSP_CAMERA | camera object reference |
| OSP_FRAMEBUFFER | framebuffer object reference |
| OSP_LIGHT | light object reference |
| OSP_MATERIAL | material object reference |
| OSP_TEXTURE | texture object reference |
| OSP_RENDERER | renderer object reference |
| OSP_WORLD | world object reference |
| OSP_GEOMETRY | geometry object reference |
| OSP_VOLUME | volume object reference |
| OSP_TRANSFER_FUNCTION | transfer function object reference |
| OSP_IMAGE_OPERATION | image operation object reference |
| OSP_STRING | C-style zero-terminated character string |
| OSP_CHAR | 8 bit signed character scalar |
| OSP_UCHAR | 8 bit unsigned character scalar |
| OSP_VEC[234]UC | ... and [234]-element vector |
| OSP_USHORT | 16 bit unsigned integer scalar |
| OSP_VEC[234]US | ... and [234]-element vector |
| OSP_INT | 32 bit signed integer scalar |
| OSP_VEC[234]I | ... and [234]-element vector |
| OSP_UINT | 32 bit unsigned integer scalar |
| OSP_VEC[234]UI | ... and [234]-element vector |
| OSP_LONG | 64 bit signed integer scalar |
| OSP_VEC[234]L | ... and [234]-element vector |
| OSP_ULONG | 64 bit unsigned integer scalar |
| OSP_VEC[234]UL | ... and [234]-element vector |
| OSP_FLOAT | 32 bit single precision floating-point scalar |
| OSP_VEC[234]F | ... and [234]-element vector |
| OSP_DOUBLE | 64 bit double precision floating-point scalar |
| OSP_BOX[1234]I | 32 bit integer box (lower + upper bounds) |
| OSP_BOX[1234]F | 32 bit single precision floating-point box (lower + upper bounds) |
| OSP_LINEAR[23]F | 32 bit single precision floating-point linear transform ([23] vectors) |
| OSP_AFFINE[23]F | 32 bit single precision floating-point affine transform (linear transform plus translation) |
| OSP_VOID_PTR | raw memory address (only found in module extensions) |

Table 3.5 – Valid named constants for `OSPDataType`.

## 3.3 Volumes

Volumes are volumetric data sets with discretely sampled values in 3D space, typically a 3D scalar field. To create a new volume object of given type `type` use

```
OSPVolume ospNewVolume(const char *type);
```

Note that OSPRay's implementation forwards `type` directly to Open VKL, allowing new Open VKL volume types to be usable within OSPRay without the need to change (or even recompile) OSPRay.

### 3.3.1 Structured Regular Volume

Structured volumes only need to store the values of the samples, because their addresses in memory can be easily computed from a 3D position. A common type of structured volumes are regular grids.

Structured regular volumes are created by passing the type string "`structuredRegular`" to `ospNewVolume`. Structured volumes are represented through an `OSPData` 3D array `data` (which may or may not be shared with the application). The voxel data must be laid out in xyz-order[2] and can be compact (best for performance) or can have a stride between voxels, specified through the `byteStride1` parameter when creating the `OSPData`. Only 1D strides are supported, additional strides between scanlines (2D, `byteStride2`) and slices (3D, `byteStride3`) are not.

[2] For consecutive memory addresses the x-index of the corresponding voxel changes the quickest.

The parameters understood by structured volumes are summarized in the table below.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | gridOrigin | $(0, 0, 0)$ | origin of the grid in object-space |
| vec3f | gridSpacing | $(1, 1, 1)$ | size of the grid cells in object-space |
| OSPData | data | | the actual voxel 3D data |

Table 3.6 – Configuration parameters for structured regular volumes.

The size of the volume is inferred from the size of the 3D array `data`, as is the type of the voxel values (currently supported are: `OSP_UCHAR`, `OSP_SHORT`, `OSP_USHORT`, `OSP_FLOAT`, and `OSP_DOUBLE`).

### 3.3.2 Structured Spherical Volume

Structured spherical volumes are also supported, which are created by passing a type string of "`structuredSpherical`" to `ospNewVolume`. The grid dimensions and parameters are defined in terms of radial distance $r$, inclination angle $\theta$, and azimuthal angle $\phi$, conforming with the ISO convention for spherical coordinate systems. The coordinate system and parameters understood by structured spherical volumes are summarized below.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | gridOrigin | $(0, 0, 0)$ | origin of the grid in units of $(r, \theta, \phi)$; angles in degrees |
| vec3f | gridSpacing | $(1, 1, 1)$ | size of the grid cells in units of $(r, \theta, \phi)$; angles in degrees |
| OSPData | data | | the actual voxel 3D data |

Table 3.7 – Configuration parameters for structured spherical volumes.

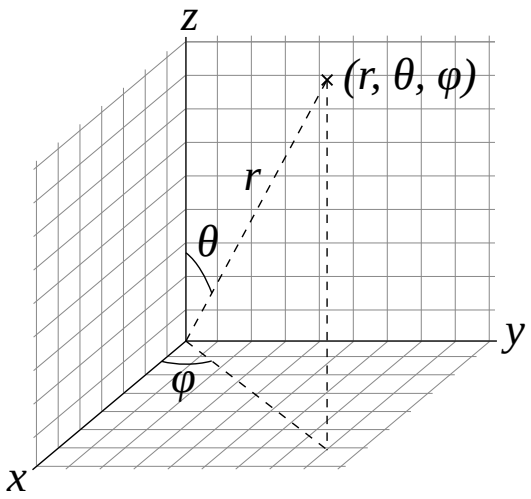


Figure 3.1 – Coordinate system of structured spherical volumes.

The dimensions $(r, \theta, \phi)$ of the volume are inferred from the size of the 3D array `data`, as is the type of the voxel values (currently supported are: `OSP_UCHAR`, `OSP_SHORT`, `OSP_USHORT`, `OSP_FLOAT`, and `OSP_DOUBLE`).

These grid parameters support flexible specification of spheres, hemispheres, spherical shells, spherical wedges, and so forth. The grid extents (computed as `[gridOrigin, gridOrigin + (dimensions - 1) * gridSpacing]`) however must be constrained such that:

- $r \geq 0$
- $0 \leq \theta \leq 180$
- $0 \leq \phi \leq 360$

### 3.3.3 Adaptive Mesh Refinement (AMR) Volume

OSPRay currently supports block-structured (Berger-Colella) AMR volumes. Volumes are specified as a list of blocks, which exist at levels of refinement in potentially overlapping regions. Blocks exist in a tree structure, with coarser refinement level blocks containing finer blocks. The cell width is equal for all blocks at the same refinement level, though blocks at a coarser level have a larger cell width than finer levels.

There can be any number of refinement levels and any number of blocks at any level of refinement. An AMR volume type is created by passing the type string "`amr`" to `ospNewVolume`.

Blocks are defined by three parameters: their bounds, the refinement level in which they reside, and the scalar data contained within each block.

Note that cell widths are defined *per refinement level*, not per block.

Lastly, note that the `gridOrigin` and `gridSpacing` parameters act just like the structured volume equivalent, but they only modify the root (coarsest level) of refinement.

In particular, OSPRay's / Open VKL's AMR implementation was designed to cover Berger-Colella [1] and Chombo [2] AMR data. The `method` parameter above determines the interpolation method used when sampling the volume.

**OSP_AMR_CURRENT** finds the finest refinement level at that cell and interpolates through this "current" level

**OSP_AMR_FINEST** will interpolate at the closest existing cell in the volume-wide finest refinement level regardless of the sample cell's level

**OSP_AMR_OCTANT** interpolates through all available refinement levels at that cell. This method avoids discontinuities at refinement level boundaries at the cost of performance

Table 3.8 – Configuration parameters for AMR volumes.

| Type | Name | Default | Description |
|---|---|---|---|
| `OSPAMRMethod` | method | `OSP_AMR_CURRENT` | `OSPAMRMethod` sampling method. Supported methods are: `OSP_AMR_CURRENT` `OSP_AMR_FINEST` `OSP_AMR_OCTANT` |
| float[] | cellWidth | NULL | array of each level's cell width |
| box3i[] | block.bounds | NULL | data array of grid sizes (in voxels) for each AMR block |
| int[] | block.level | NULL | array of each block's refinement level |
| OSPData[] | block.data | NULL | data array of OSPData containing the actual scalar voxel data, only `OSP_FLOAT` is supported as `OSPDataType` |
| vec3f | gridOrigin | $(0, 0, 0)$ | origin of the grid in world-space |
| vec3f | gridSpacing | $(1, 1, 1)$ | size of the grid cells in world-space |

Details and more information can be found in the publication for the implementation [3].


1. M.J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics." Journal of Computational Physics 82.1 (1989): 64-84. DOI: 10.1016/0021-9991(89)90035-1
2. M. Adams, P. Colella, D.T. Graves, J.N. Johnson, N.D. Keen, T.J. Ligocki, D.F. Martin. P.W. McCorquodale, D. Modiano. P.O. Schwartz, T.D. Sternberg, and B. Van Straalen, "Chombo Software Package for AMR Applications – Design Document", Lawrence Berkeley National Laboratory Technical Report LBNL-6616E.
3. I. Wald, C. Brownlee, W. Usher, and A. Knoll, "CPU volume rendering of adaptive mesh refinement data". SIGGRAPH Asia 2017 Symposium on Visualization – SA '17, 18(8), 1–8. DOI: 10.1145/3139295.3139305


### 3.3.4 Unstructured Volume

Unstructured volumes can have their topology and geometry freely defined. Geometry can be composed of tetrahedral, hexahedral, wedge or pyramid cell types. The data format used is compatible with VTK and consists of multiple arrays: vertex positions and values, vertex indices, cell start indices, cell types, and cell values. An unstructured volume type is created by passing the type string "`unstructured`" to `ospNewVolume`.

Sampled cell values can be specified either per-vertex (`vertex.data`) or per-cell (`cell.data`). If both arrays are set, `cell.data` takes precedence.

Similar to a mesh, each cell is formed by a group of indices into the vertices. For each vertex, the corresponding (by array index) data value will be used for sampling when rendering, if specified. The index order for a tetrahedron is the same as `VTK_TETRA`: bottom triangle counterclockwise, then the top vertex.

For hexahedral cells, each hexahedron is formed by a group of eight indices into the vertices and data values. Vertex ordering is the same as `VTK_HEXAHEDRON`: four bottom vertices counterclockwise, then top four counterclockwise.

For wedge cells, each wedge is formed by a group of six indices into the vertices and data values. Vertex ordering is the same as `VTK_WEDGE`: three bottom vertices counterclockwise, then top three counterclockwise.

For pyramid cells, each cell is formed by a group of five indices into the vertices and data values. Vertex ordering is the same as `VTK_PYRAMID`: four bottom vertices counterclockwise, then the top vertex.

To maintain VTK data compatibility, the `index` array may be specified with cell sizes interleaved with vertex indices in the following format: $n, id_1, ..., id_n, m, id_1, ..., id_m$. This alternative `index` array layout can be enabled through the `indexPrefixed` flag (in which case, the `cell.type` parameter must be omitted).

Table 3.9 – Configuration parameters for unstructured volumes.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f[] | vertex.position | | data array of vertex positions |
| float[] | vertex.data | | data array of vertex data values to be sampled |
| uint32[] / uint64[] | index | | data array of indices (into the vertex array(s)) that form cells |
| bool | indexPrefixed | false | indicates that the `index` array is compatible to VTK, where the indices of each cell are prefixed with the number of vertices |
| uint32[] / uint64[] | cell.index | | data array of locations (into the index array), specifying the first index of each cell |
| float[] | cell.data | | data array of cell data values to be sampled |
| uint8[] | cell.type | | data array of cell types (VTK compatible), only set if `indexPrefixed = false` false. Supported types are: `OSP_TETRAHEDRON` `OSP_HEXAHEDRON` `OSP_WEDGE` `OSP_PYRAMID` |
| bool | hexIterative | false | hexahedron interpolation method, defaults to fast non-iterative version which could have rendering inaccuracies may appear if hex is not parallelepiped |
| bool | precomputedNormals | false | whether to accelerate by precomputing, at a cost of 12 bytes/face |

### 3.3.5 VDB Volume

VDB volumes implement a data structure that is very similar to the data structure outlined in Museth [1], they are created by passing the type string "`vdb`" to `ospNewVolume`.

The data structure is a hierarchical regular grid at its core: Nodes are regular grids, and each grid cell may either store a constant value (this is called a tile), or child pointers. Nodes in VDB trees are wide: Nodes on the first level have a resolution of $32^3$ voxels, on the next level $16^3$, and on the leaf level $8^3$ voxels. All nodes on a given level have the same resolution. This makes it easy to find the node containing a coordinate using shift operations (see [1]). VDB leaf nodes are implicit in OSPRay / Open VKL: they are stored as pointers to user-provided data.

VDB volumes interpret input data as constant cells (which are then potentially filtered). This is in contrast to `structuredRegular` volumes, which have a vertex-centered interpretation.

The VDB implementation in OSPRay / Open VKL follows the following goals:

- Efficient data structure traversal on vector architectures.
- Enable the use of industry-standard `.vdb` files created through the OpenVDB library.
- Compatibility with OpenVDB on a leaf data level, so that `.vdb` file may be loaded with minimal overhead.

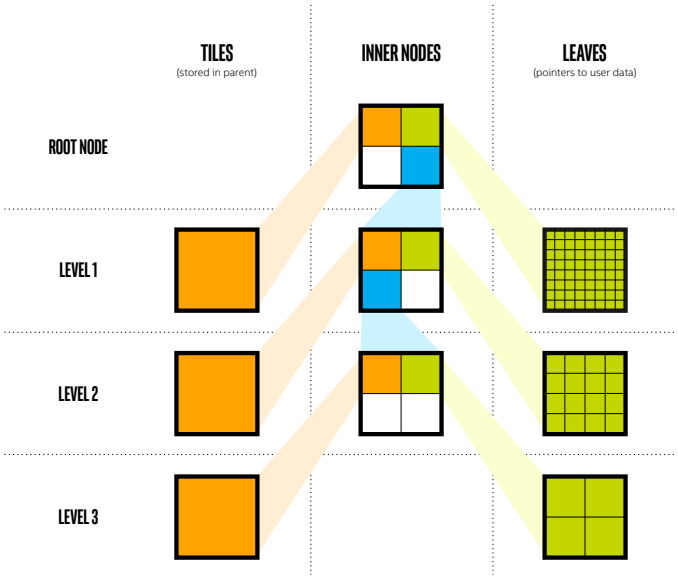


Figure 3.2 – Topology of VDB volumes.

VDB volumes have the following parameters:

Table 3.10 – Configuration parameters for VDB volumes.

| Type | Name | Description |
|---|---|---|
| int | maxIteratorDepth | do not descend further than to this depth during interval iteration, the maximum value and the default is 3 |
| int | maxSamplingDepth | do not descend further than to this depth during sampling, the maximum value and the default is 3 |
| uint32[] | node.level | level on which each input node exists, may be 1, 2 or 3 (levels are counted from the root level = 0 down) |
| vec3i[] | node.origin | the node origin index (per input node) |
| OSPData[] | node.data | data arrays with the node data (per input node). Nodes that are tiles are expected to have single-item arrays. Leaf-nodes with grid data expected to have compact 3D arrays in zyx layout (z changes most quickly) with the correct number of voxels for the `level`. Only `OSP_FLOAT` is supported as field `OSPDataType`. |
| int | filter | filter used for reconstructing the field, default is `OSP_VOLUME_FILTER_TRILINEAR`, alternatively `OSP_VOLUME_FILTER_NEAREST`. |
| int | gradientFilter | filter used for reconstructing the field during gradient computations, default same as `filter` |

1. Museth, K. VDB: High-Resolution Sparse Volumes with Dynamic Topology. ACM Transactions on Graphics 32(3), 2013. DOI: 10.1145/2487228.2487235

### 3.3.6 Particle Volume

Particle volumes consist of a set of points in space. Each point has a position, a radius, and a weight typically associated with an attribute. Particle volumes are created by passing the type string "`particle`" to `ospNewVolume`.

A radial basis function defines the contribution of that particle. Currently, we use the Gaussian radial basis function

$$\phi(P) = w \exp\left(-\frac{(P-p)^2}{2r^2}\right),$$

where $P$ is the particle position, $p$ is the sample position, $r$ is the radius and $w$ is the weight. At each sample, the scalar field value is then computed as the sum of each radial basis function $\phi$, for each particle that overlaps it.

The OSPRay / Open VKL implementation is similar to direct evaluation of samples in Reda et al. [2]. It uses an Embree-built BVH with a custom traversal, similar to the method in [1].

Table 3.11 – Configuration parameters for particle volumes.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f[] | particle.position | | data array of particle positions |
| float[] | particle.radius | | data array of particle radii |
| float[] | particle.weight | NULL | optional data array of particle weights, specifying the height of the kernel. |
| float | radiusSupportFactor | 3.0 | The multiplier of the particle radius required for support. Larger radii ensure smooth results at the cost of performance. In the Gaussian kernel, the radius is one standard deviation ($\sigma$), so a value of 3 corresponds to $3\sigma$. |
| float | clampMaxCumulativeValue | 0 | The maximum cumulative value possible, set by user. All cumulative values will be clamped to this, and further traversal (RBF summation) of particle contributions will halt when this value is reached. A value of zero or less turns this off. |
| bool | estimateValueRanges | true | Enable heuristic estimation of value ranges which are used in internal acceleration structures as well as for determining the volume's overall value range. When set to `false`, the user *must* specify `clampMaxCumulativeValue`, and all value ranges will be assumed [0, `clampMaxCumulativeValue`]. Disabling this switch may improve volume commit time, but will make volume rendering less efficient. |

1. A. Knoll, I. Wald, P. Navratil, A. Bowen, K. Reda, M.E., Papka, and K. Gaither, "RBF Volume Ray Casting on Multicore and Manycore CPUs", 2014, Computer Graphics Forum, 33: 71–80. doi:10.1111/cgf.12363

2. K. Reda, A. Knoll, K. Nomura, M. E. Papka, A. E. Johnson and J. Leigh, "Visualizing large-scale atomistic simulations in ultra-resolution immersive environments", 2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), Atlanta, GA, 2013, pp. 59–65.

### 3.3.7 Transfer Function

Transfer functions map the scalar values of volumes to color and opacity and thus they can be used to visually emphasize certain features of the volume. To create a new transfer function of given type `type` use

```
OSPTransferFunction ospNewTransferFunction(const char *type);
```

The returned handle can be assigned to a volumetric model (described below) as parameter "`transferFunction`" using `ospSetObject`.

One type of transfer function that is supported by OSPRay is the linear transfer function, which interpolates between given equidistant colors and opacities. It is create by passing the string "`piecewiseLinear`" to `ospNewTransferFunction` and it is controlled by these parameters:

The arrays `color` and `opacity` can be of different length.

| Type | Name | Description |
|---|---|---|
| vec3f[] | color | data array of RGB colors |
| float[] | opacity | data array of opacities |
| vec2f | valueRange | domain (scalar range) this function maps from |

Table 3.12 – Parameters accepted by the linear transfer function.

### 3.3.8 VolumetricModels

Volumes in OSPRay are given volume rendering appearance information through VolumetricModels. This decouples the physical representation of the volume (and possible acceleration structures it contains) to rendering-specific parameters (where more than one set may exist concurrently). To create a volume instance, call

```
OSPVolumetricModel ospNewVolumetricModel(OSPVolume volume);
```

Table 3.13 – Parameters understood by VolumetricModel.

| Type | Name | Default | Description |
|---|---|---|---|
| OSPTransferFunction | transferFunction | | transfer function to use |
| float | densityScale | 1.0 | makes volumes uniformly thinner or thicker |
| float | anisotropy | 0.0 | anisotropy of the (Henyey-Greenstein) phase function in [-1, 1] (path tracer only), default to isotropic scattering |

## 3.4 Geometries

Geometries in OSPRay are objects that describe intersectable surfaces. To create a new geometry object of given type `type` use

```
OSPGeometry ospNewGeometry(const char *type);
```

Note that in the current implementation geometries are limited to a maximum of $2^{32}$ primitives.

### 3.4.1 Mesh

A mesh consisting of either triangles or quads is created by calling `ospNewGeometry` with type string "`mesh`". Once created, a mesh recognizes the following parameters:

| Type | Name | Description |
|---|---|---|
| vec3f[] | vertex.position | data array of vertex positions |
| vec3f[] | vertex.normal | data array of vertex normals |
| vec4f[] / vec3f[] | vertex.color | data array of vertex colors (RGBA/RGB) |
| vec2f[] | vertex.texcoord | data array of vertex texture coordinates |
| vec3ui[] / vec4ui[] | index | data array of (either triangle or quad) indices (into the vertex array(s)) |

Table 3.14 – Parameters defining a mesh geometry.

The data type of index arrays differentiates between the underlying geometry, triangles are used for a index with `vec3ui` type and quads for `vec4ui` type. Quads are internally handled as a pair of two triangles, thus mixing triangles and quads is supported by encoding some triangle as a quad with the last two vertex indices being identical (`w=z`).

The `vertex.position` and `index` arrays are mandatory to create a valid mesh.

### 3.4.2 Subdivision

A mesh consisting of subdivision surfaces, created by specifying a geometry of type "`subdivision`". Once created, a subdivision recognizes the following parameters:

Table 3.15 – Parameters defining a Subdivision geometry.

| Type | Name | Description |
|---|---|---|
| vec3f[] | vertex.position | data array of vertex positions |
| vec4f[] | vertex.color | optional data array of vertex colors (RGBA) |
| vec2f[] | vertex.texcoord | optional data array of vertex texture coordinates |
| float | level | global level of tessellation, default 5 |
| uint[] | index | data array of indices (into the vertex array(s)) |
| float[] | index.level | optional data array of per-edge levels of tessellation, overrides global level |
| uint[] | face | optional data array holding the number of indices/edges (3 to 15) per face, defaults to 4 (a pure quad mesh) |
| vec2i[] | edgeCrease.index | optional data array of edge crease indices |
| float[] | edgeCrease.weight | optional data array of edge crease weights |
| uint[] | vertexCrease.index | optional data array of vertex crease indices |
| float[] | vertexCrease.weight | optional data array of vertex crease weights |
| uchar | mode | subdivision edge boundary mode, supported modes are:<br>`OSP_SUBDIVISION_NO_BOUNDARY`<br>`OSP_SUBDIVISION_SMOOTH_BOUNDARY` (default)<br>`OSP_SUBDIVISION_PIN_CORNERS`<br>`OSP_SUBDIVISION_PIN_BOUNDARY`<br>`OSP_SUBDIVISION_PIN_ALL` |

The `vertex` and `index` arrays are mandatory to create a valid subdivision surface. If no `face` array is present then a pure quad mesh is assumed (the number of indices must be a multiple of 4). Optionally supported are edge and vertex creases.

### 3.4.3 Spheres

A geometry consisting of individual spheres, each of which can have an own radius, is created by calling `ospNewGeometry` with type string "`sphere`". The spheres will not be tessellated but rendered procedurally and are thus perfectly round. To allow a variety of sphere representations in the application this geometry allows a flexible way of specifying the data of center position and radius within a data array:

Table 3.16 – Parameters defining a spheres geometry.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f[] | sphere.position | | data array of center positions |
| float[] | sphere.radius | NULL | optional data array of the per-sphere radius |
| vec2f[] | sphere.texcoord | NULL | optional data array of texture coordinates (constant per sphere) |
| float | radius | 0.01 | default radius for all spheres (if `sphere.radius` is not set) |

### 3.4.4 Curves

A geometry consisting of multiple curves is created by calling `ospNewGeometry` with type string "`curve`". The parameters defining this geometry are listed in the table below.

Table 3.17 – Parameters defining a curves geometry.

| Type | Name | Description |
|---|---|---|
| vec4f[] | vertex.position_radius | data array of vertex position and per-vertex radius |
| vec3f[] | vertex.position | data array of vertex position |
| float | radius | global radius of all curves (if per-vertex radius is not used), default 0.01 |
| vec2f[] | vertex.texcoord | data array of per-vertex texture coordinates |
| vec4f[] | vertex.color | data array of corresponding vertex colors (RGBA) |
| vec3f[] | vertex.normal | data array of curve normals (only for "ribbon" curves) |
| vec4f[] | vertex.tangent | data array of curve tangents (only for "hermite" curves) |
| uint32[] | index | data array of indices to the first vertex or tangent of a curve segment |
| uchar | type | `OSPCurveType` for rendering the curve. Supported types are:<br>`OSP_FLAT`<br>`OSP_ROUND`<br>`OSP_RIBBON` |
| uchar | basis | `OSPCurveBasis` for defining the curve. Supported bases are:<br>`OSP_LINEAR`<br>`OSP_BEZIER`<br>`OSP_BSPLINE`<br>`OSP_HERMITE`<br>`OSP_CATMULL_ROM` |

Depending upon the specified data type of vertex positions, the curves will be implemented Embree curves or assembled from rounded and linearly-connected segments.

Positions in `vertex.position_radius` format supports per-vertex varying radii with data type `vec4f[]` and instantiate Embree curves internally for the relevant type/basis mapping.

If a constant `radius` is used and positions are specified in a `vec3f[]` type of `vertex.position` format, then type/basis defaults to `OSP_ROUND` and `OSP_LINEAR` (this is the fastest and most memory efficient mode). Implementation is with round linear segments where each segment corresponds to a link between two vertices.

The following section describes the properties of different curve basis' and how they use the data provided in data buffers:

**OSP_LINEAR** 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. The curve goes through all control points listed in the vertex buffer.

**OSP_BEZIER** The indices point to the first of 4 consecutive control points in the vertex buffer. 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.

**OSP_BSPLINE** The indices point to the first of 4 consecutive control points in the vertex buffer. This basis is not interpolating, thus the curve does in general not go through any of the control points directly. Using this basis, 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 make this basis a good choice to construct continuous multi-segment curves, as memory consumption can be kept minimal.

**OSP_HERMITE** It is necessary to have both vertex buffer and tangent buffer for using this 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. 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.

**OSP_CATMULL_ROM** The indices point to the first of 4 consecutive control points in the vertex buffer. If $(p0, p1, p2, p3)$ represent the points then this basis goes through $p1$ and $p2$, with tangents as $(p2-p0)/2$ and $(p3-p1)/2$.

The following section describes the properties of different curve types' and how they define the geometry of a curve:

**OSP_FLAT** This type enables faster rendering as the curve is rendered as a connected sequence of ray facing quads.

**OSP_ROUND** This type enables rendering a real geometric surface for the curve which allows closeup views. This mode renders a sweep surface by sweeping a varying radius circle tangential along the curve.

**OSP_RIBBON** The type enables normal orientation of the curve and requires a normal buffer be specified along with vertex buffer. The curve is rendered as a flat band whose center approximately follows the provided vertex buffer and whose normal orientation approximately follows the provided normal buffer.

### 3.4.5 Boxes

OSPRay can directly render axis-aligned bounding boxes without the need to convert them to quads or triangles. To do so create a boxes geometry by calling `ospNewGeometry` with type string "`box`".

| Type | Name | Description |
|---|---|---|
| box3f[] | box | data array of boxes |

Table 3.18 – Parameters defining a boxes geometry.

### 3.4.6 Planes

OSPRay can directly render planes defined by plane equation coefficients in its implicit form $ax+by+cz+d = 0$. By default planes are infinite but their extents

can be limited by defining optional bounding boxes. A planes geometry can be created by calling `ospNewGeometry` with type string "`plane`".

| Type | Name | Descript ion |
|---|---|---|
| vec4f[] | plane.coefficients | data array of plane coefficients $(a, b, c, d)$ |
| box3f[] | plane.bounds | optional data array of bounding boxes |

Table 3.19 – Parameters defining a planes geometry.

### 3.4.7 Isosurfaces

OSPRay can directly render multiple isosurfaces of a volume without first tessellating them. To do so create an isosurfaces geometry by calling `ospNewGeometry` with type string "`isosurface`". The appearance information of the surfaces is set through the Geometric Model. Per-isosurface colors can be set by passing per-primitive colors to the Geometric Model, in order of the isosurface array.

| Type | Name | Description |
|---|---|---|
| float | isovalue | single isovalues |
| float[] | isovalue | data array of isovalues |
| OSPVolume | volume | handle of the Volume to be isosurfaced |

Table 3.20 – Parameters defining an isosurfaces geometry.

### 3.4.8 GeometricModels

Geometries are matched with surface appearance information through GeometricModels. These take a geometry, which defines the surface representation, and applies either full-object or per-primitive color and material information. To create a geometric model, call

```
OSPGeometricModel ospNewGeometricModel(OSPGeometry geometry);
```

Color and material are fetched with the primitive ID of the hit (clamped to the valid range, thus a single color or material is fine), or mapped first via the `index` array (if present). All parameters are optional, however, some renderers (notably the path tracer) require a material to be set. Materials are either handles of `OSPMaterial`, or indices into the `material` array on the renderer, which allows to build a world which can be used by different types of renderers.

An `invertNormals` flag allows to invert (shading) normal vectors of the rendered geometry. That is particularly useful for clipping. By changing normal vectors orientation one can control whether inside or outside of the clipping geometry is being removed. For example, a clipping geometry with normals oriented outside clips everything what's inside.

## 3.5 Lights

To create a new light source of given type `type` use

```
OSPLight ospNewLight(const char *type);
```

All light sources accept the following parameters:
The following light types are supported by most OSPRay renderers.

Table 3.21 – Parameters understood by GeometricModel.

| Type | Name | Description |
|---|---|---|
| OSPMaterial / uint32 | material | optional material applied to the geometry, may be an index into the `material` parameter on the renderer (if it exists) |
| vec4f | color | optional color assigned to the geometry |
| OSPMaterial[] / uint32[] | material | optional data array of (per-primitive) materials, may be an index into the `material` parameter on the renderer (if it exists) |
| vec4f[] | color | optional data array of (per-primitive) colors |
| uint8[] | index | optional data array of per-primitive indices into `color` and `material` |
| bool | invertNormals | inverts all shading normals (Ns), default false |

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | color | white | color of the light |
| float | intensity | 1 | intensity of the light (a factor) |
| bool | visible | true | whether the light can be directly seen |

Table 3.22 – Parameters accepted by all lights.

### 3.5.1 Directional Light / Distant Light

The distant light (or traditionally the directional light) is thought to be far away (outside of the scene), thus its light arrives (almost) as parallel rays. It is created by passing the type string "`distant`" to `ospNewLight`. In addition to the general parameters understood by all lights the distant light supports the following special parameters:

| Type | Name | Description |
|---|---|---|
| vec3f | direction | main emission direction of the distant light |
| float | angularDiameter | apparent size (angle in degree) of the light |

Table 3.23 – Special parameters accepted by the distant light.

Setting the angular diameter to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the path tracer). For instance, the apparent size of the sun is about 0.53°.

### 3.5.2 Point Light / Sphere Light

The sphere light (or the special case point light) is a light emitting uniformly in all directions from the surface toward the outside. It does not emit any light toward the inside of the sphere. It is created by passing the type string "`sphere`" to `ospNewLight`. In addition to the general parameters understood by all lights the sphere light supports the following special parameters:

| Type | Name | Description |
|---|---|---|
| vec3f | position | the center of the sphere light, in world-space |
| float | radius | the size of the sphere light |

Table 3.24 – Special parameters accepted by the sphere light.

Setting the radius to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the path tracer).

### 3.5.3 Spotlight / Photometric Light

The spotlight is a light emitting into a cone of directions. It is created by passing the type string "spot" to ospNewLight. In addition to the general parameters understood by all lights the spotlight supports the special parameters listed in the table.

Table 3.25 – Special parameters accepted by the spotlight.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | position | $(0, 0, 0)$ | the center of the spotlight, in world-space |
| vec3f | direction | $(0, 0, 1)$ | main emission direction of the spot |
| float | openingAngle | 180 | full opening angle (in degree) of the spot; outside of this cone is no illumination |
| float | penumbraAngle | 5 | size (angle in degree) of the "penumbra", the region between the rim (of the illumination cone) and full intensity of the spot; should be smaller than half of openingAngle |
| float | radius | 0 | the size of the spotlight, the radius of a disk with normal direction |
| float | innerRadius | 0 | in combination with radius turns the disk into a ring |
| float[] | intensityDistribution | | luminous intensity distribution for photometric lights; can be 2D for asymmetric illumination; values are assumed to be uniformly distributed |
| vec3f | c0 | | orientation, i.e., direction of the C0-(half)plane (only needed if illumination via intensityDistribution is asymmetric) |

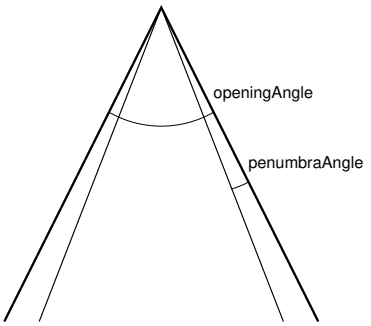


Figure 3.3 – Angles used by the spotlight.

Setting the radius to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the path tracer). Additionally setting the inner radius will result in a ring instead of a disk emitting the light.

Measured light sources (IES, EULUMDAT, ...) are supported by providing an intensityDistribution data array to modulate the intensity per direction. The mapping is using the C-$\gamma$ coordinate system (see also below figure): the values of the first (or only) dimension of intensityDistribution are uniformly mapped to $\gamma$ in $[0–\pi]$; the first intensity value to 0, the last value to $\pi$, thus at least two values need to be present. If the array has a second dimension then the intensities are not rotational symmetric around direction, but are accordingly mapped to the C-halfplanes in $[0–2\pi]$; the first "row" of values to 0 and $2\pi$, the other rows such that they have uniform distance to its neighbors. The orientation of the C0-plane is specified via c0.

### 3.5.4 Quad Light

The quad[3] light is a planar, procedural area light source emitting uniformly on one side into the half-space. It is created by passing the type string "quad" to

[3] actually a parallelogram

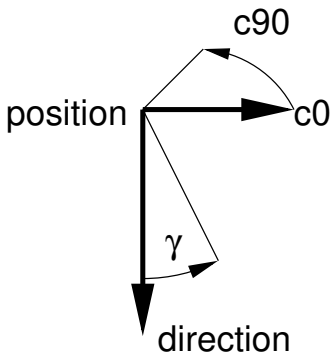


Figure 3.4 – C-γ coordinate system for the mapping of `intensityDistribution` to the spotlight.

`ospNewLight`. In addition to the general parameters understood by all lights the quad light supports the following special parameters:

| Type | Name | Description |
|---|---|---|
| vec3f | position | world-space position of one vertex of the quad light |
| vec3f | edge1 | vector to one adjacent vertex |
| vec3f | edge2 | vector to the other adjacent vertex |

Table 3.26 – Special parameters accepted by the quad light.

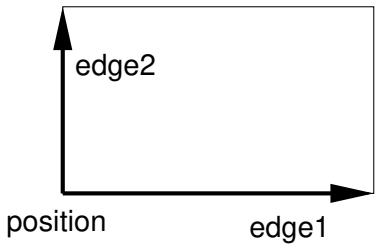


Figure 3.5 – Defining a quad light which emits toward the reader.

The emission side is determined by the cross product of `edge1×edge2`. Note that only renderers that use stochastic sampling (like the path tracer) will compute soft shadows from the quad light. Other renderers will just sample the center of the quad light, which results in hard shadows.

### 3.5.5 HDRI Light

The HDRI light is a textured light source surrounding the scene and illuminating it from infinity. It is created by passing the type string "`hdri`" to `ospNewLight`. In addition to the general parameters the HDRI light supports the following special parameters:

Table 3.27 – Special parameters accepted by the HDRI light.

| Type | Name | Description |
|---|---|---|
| vec3f | up | up direction of the light in world-space |
| vec3f | direction | direction to which the center of the texture will be mapped to (analog to panoramic camera) |
| OSPTexture | map | environment map in latitude / longitude format |

Note that the currently only the path tracer supports the HDRI light.

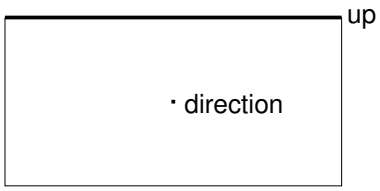


Figure 3.6 – Orientation and Mapping of an HDRI Light.

### 3.5.6 Ambient Light

The ambient light surrounds the scene and illuminates it from infinity with constant radiance (determined by combining the parameters `color` and `intensity`). It is created by passing the type string "`ambient`" to `ospNewLight`.

Note that the SciVis renderer uses ambient lights to control the color and intensity of the computed ambient occlusion (AO).

### 3.5.7 Sun-Sky Light

The sun-sky light is a combination of a `distant` light for the sun and a procedural `hdri` light for the sky. It is created by passing the type string "`sunSky`" to `ospNewLight`. The sun-sky light surrounds the scene and illuminates it from infinity and can be used for rendering outdoor scenes. The radiance values are calculated using the Hošek-Wilkie sky model and solar radiance function. In addition to the general parameters the following special parameters are supported:

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | up | $(0, 1, 0)$ | zenith of sky in world-space |
| vec3f | direction | $(0, -1, 0)$ | main emission direction of the sun |
| float | turbidity | 3 | atmospheric turbidity due to particles, in [1–10] |
| float | albedo | 0.3 | ground reflectance, in [0–1] |

Table 3.28 – Special parameters accepted by the `sunSky` light.

The lowest elevation for the sun is restricted to the horizon.

### 3.5.8 Emissive Objects

The path tracer will consider illumination by geometries which have a light emitting material assigned (for example the Luminous material).

## 3.6 Scene Hierarchy

### 3.6.1 Groups

Groups in OSPRay represent collections of GeometricModels and VolumetricModels which share a common local-space coordinate system. To create a group call

```
OSPGroup ospNewGroup();
```

Groups take arrays of geometric models, volumetric models and clipping geometric models, but they are optional. In other words, there is no need to create empty arrays if there are no geometries or volumes in the group.

By adding `OSPGeometricModels` to the `clippingGeometry` array a clipping geometry feature is enabled. Geometries assigned to this parameter will be used as clipping geometries. Any supported geometry can be used for clipping. The

only requirement is that it has to distinctly partition space into clipping and non-clipping one. These include: spheres, boxes, infinite planes, closed meshes, closed subdivisions and curves. All geometries and volumes assigned to `geometry` or `volume` will be clipped. Use of clipping geometry that is not closed (or infinite) will result in rendering artifacts. User can decide which part of space is clipped by changing shading normals orientation with the `invertNormals` flag of the GeometricModel. When more than single clipping geometry is defined all clipping areas will be "added" together – an union of these areas will be applied.

Table 3.29 – Parameters understood by groups.

| Type | Name | Default | Description |
|---|---|---|---|
| OSPGeometricModel[] | geometry | NULL | data array of GeometricModels |
| OSPVolumetricModel[] | volume | NULL | data array of VolumetricModels |
| OSPGeometricModel[] | clippingGeometry | NULL | data array of GeometricModels used for clipping |
| bool | dynamicScene | false | use RTC_SCENE_DYNAMIC flag (faster BVH build, slower ray traversal), otherwise uses RTC_SCENE_STATIC flag (faster ray traversal, slightly slower BVH build) |
| bool | compactMode | false | tell Embree to use a more compact BVH in memory by trading ray traversal performance |
| bool | robustMode | false | tell Embree to enable more robust ray intersection code paths (slightly slower) |

Note that groups only need to re re-committed if a geometry or volume changes (surface/scalar field representation). Appearance information on `OSPGeometricModel` and `OSPVolumetricModel` can be changed freely, as internal acceleration structures do not need to be reconstructed.

### 3.6.2 Instances

Instances in OSPRay represent a single group's placement into the world via a transform. To create and instance call

```
OSPInstance ospNewInstance(OSPGroup);
```

Table 3.30 – Parameters understood by instances.

| Type | Name | Default | Description |
|---|---|---|---|
| affine3f | xfm | identity | world-space transform for all attached geometries and volumes |

### 3.6.3 World

Worlds are a container of scene data represented by instances. To create an (empty) world call

```
OSPWorld ospNewWorld();
```

Objects are placed in the world through an array of instances. Similar to groups, the array of instances is optional: there is no need to create empty arrays if there are no instances (though there will be nothing to render).

Applications can query the world (axis-aligned) bounding box after the world has been committed. To get this information, call

```
OSPBounds ospGetBounds(OSPObject);
```

The result is returned in the provided `OSPBounds`[4] struct:

[4] `OSPBounds` has essentially the same layout as the `OSP_BOX3F` `OSPDataType`.

```
typedef struct {
    float lower[3];
    float upper[3];
} OSPBounds;
```

This call can also take `OSPGroup` and `OSPInstance` as well: all other object types will return an empty bounding box.

Finally, Worlds can be configured with parameters for making various feature/performance trade-offs (similar to groups).

Table 3.31 – Parameters understood by worlds.

| Type | Name | Default | Description |
|---|---|---|---|
| OSPInstance[] | instance | NULL | data array with handles of the instances |
| OSPLight[] | light | NULL | data array with handles of the lights |
| bool | dynamicScene | false | use RTC_SCENE_DYNAMIC flag (faster BVH build, slower ray traversal), otherwise uses RTC_SCENE_STATIC flag (faster ray traversal, slightly slower BVH build) |
| bool | compactMode | false | tell Embree to use a more compact BVH in memory by trading ray traversal performance |
| bool | robustMode | false | tell Embree to enable more robust ray intersection code paths (slightly slower) |

## 3.7 Renderers

A renderer is the central object for rendering in OSPRay. Different renderers implement different features and support different materials. To create a new renderer of given type `type` use

```
OSPRenderer ospNewRenderer(const char *type);
```

General parameters of all renderers are

OSPRay's renderers support a feature called adaptive accumulation, which accelerates progressive rendering by stopping the rendering and refinement of image regions that have an estimated variance below the `varianceThreshold`. This feature requires a framebuffer with an `OSP_FB_VARIANCE` channel.

Per default the background of the rendered image will be transparent black, i.e., the alpha channel holds the opacity of the rendered objects. This eases transparency-aware blending of the image with an arbitrary background image by the application. The parameter `backgroundColor` or `map_backplate` can be used to already blend with a constant background color or backplate texture, respectively, (and alpha) during rendering.

OSPRay renderers support depth composition with images of other renderers, for example to incorporate help geometries of a 3D UI that were rendered with OpenGL. The screen-sized texture `map_maxDepth` must have format `OSP_TEXTURE_R32F` and flag `OSP_TEXTURE_FILTER_NEAREST`. The fetched values are used to limit the distance of primary rays, thus objects of other renderers can hide objects rendered by OSPRay.

OSPRay supports antialiasing in image space by using pixel filters, which are centered around the center of a pixel. The size $w \times w$ of the filter depends on

Table 3.32 – Parameters understood by all renderers.

| Type | Name | Default | Description |
|---|---|---|---|
| int | pixelSamples | 1 | samples per pixel |
| int | maxPathLength | 20 | maximum ray recursion depth |
| float | minContribution | 0.001 | sample contributions below this value will be neglected to speedup rendering |
| float | varianceThreshold | 0 | threshold for adaptive accumulation |
| float / vec3f / vec4f | backgroundColor | black, transparent | background color and alpha (RGBA), if no `map_backplate` is set |
| OSPTexture | map_backplate | | optional texture image used as background (use texture type `texture2d`) |
| OSPTexture | map_maxDepth | | optional screen-sized float texture with maximum far distance per pixel (use texture type `texture2d`) |
| OSPMaterial[] | material | | optional data array of materials which can be indexed by a GeometricModel's `material` parameter |
| uchar | pixelFilter | `OSP_PIXELFILTER_GAUSS` | `OSPPixelFilterType` to select the pixel filter used by the renderer for antialiasing. Possible pixel filters are listed below. |

the selected filter type. The types of supported pixel filters are defined by the `OSPPixelFilterType` enum and can be set using the `pixelFilter` parameter.

Table 3.33 – Pixel filter types supported by OSPRay for antialiasing in image space.

| Name | Description |
|---|---|
| OSP_PIXELFILTER_POINT | a point filter only samples the center of the pixel, therefore the filter width is $w = 0$ |
| OSP_PIXELFILTER_BOX | a uniform box filter with a width of $w = 1$ |
| OSP_PIXELFILTER_GAUSS | a truncated, smooth Gaussian filter with a standard deviation of $\sigma = 0.5$ and a filter width of $w = 3$ |
| OSP_PIXELFILTER_MITCHELL | the Mitchell-Netravali filter with a width of $w = 4$ |
| OSP_PIXELFILTER_BLACKMAN_HARRIS | the Blackman-Harris filter with a width of $w = 3$ |

### 3.7.1 SciVis Renderer

The SciVis renderer is a fast ray tracer for scientific visualization which supports volume rendering and ambient occlusion (AO). It is created by passing the type string "`scivis`" to `ospNewRenderer`. In addition to the general parameters understood by all renderers, the SciVis renderer supports the following parameters:

Note that the intensity (and color) of AO is deduced from an ambient light in the `lights` array.[5] If `aoSamples` is zero (the default) then ambient lights cause ambient illumination (without occlusion).

[5] If there are multiple ambient lights then their contribution is added

### 3.7.2 Ambient Occlusion Renderer

This renderer supports only a subset of the features of the SciVis renderer to gain performance. As the name suggest its main shading method is ambient occlusion

Table 3.34 – Special parameters understood by the SciVis renderer.

| Type | Name | Default | Description |
|---|---|---|---|
| bool | shadows | false | whether to compute (hard) shadows |
| int | aoSamples | 0 | number of rays per sample to compute ambient occlusion |
| float | aoDistance | $10^{20}$ | maximum distance to consider for ambient occlusion |
| float | volumeSamplingRate | 1 | sampling rate for volumes |

(AO), lights are *not* considered at all and , Volume rendering is supported. The Ambient Occlusion renderer is created by passing the type string "`ao`" to `ospNewRenderer`. In addition to the general parameters understood by all renderers the following parameters are supported as well:

Table 3.35 – Special parameters understood by the Ambient Occlusion renderer.

| Type | Name | Default | Description |
|---|---|---|---|
| int | aoSamples | 1 | number of rays per sample to compute ambient occlusion |
| float | aoDistance | $10^{20}$ | maximum distance to consider for ambient occlusion |
| float | aoIntensity | 1 | ambient occlusion strength |
| float | volumeSamplingRate | 1 | sampling rate for volumes |

### 3.7.3 Path Tracer

The path tracer supports soft shadows, indirect illumination and realistic materials. This renderer is created by passing the type string "`pathtracer`" to `ospNewRenderer`. In addition to the general parameters understood by all renderers the path tracer supports the following special parameters:

Table 3.36 – Special parameters understood by the path tracer.

| Type | Name | Default | Description |
|---|---|---|---|
| int | lightSamples | all | number of random light samples per path vertex, per default all light sources are sampled |
| bool | geometryLights | true | whether geometries with an emissive material (e.g., Luminous) illuminate the scene |
| int | roulettePathLength | 5 | ray recursion depth at which to start Russian roulette termination |
| float | maxContribution | $\infty$ | samples are clamped to this value before they are accumulated into the framebuffer |

The path tracer requires that materials are assigned to geometries, otherwise surfaces are treated as completely black.

The path tracer supports volumes with multiple scattering. The scattering albedo can be specified using the transfer function. Extinction is assumed to be spectrally constant.

### 3.7.4 Materials

Materials describe how light interacts with surfaces, they give objects their distinctive look. To let the given renderer create a new material of given type `type` call

```
OSPMaterial ospNewMaterial(const char *renderer_type, const char *material_type);
```

The returned handle can then be used to assign the material to a given geometry with

```
void ospSetObject(OSPGeometricModel, "material", OSPMaterial);
```

#### 3.7.4.1 OBJ Material

The OBJ material is the workhorse material supported by both the SciVis renderer and the path tracer (the Ambient Occlusion renderer only uses the `kd` and `d` parameter). It offers widely used common properties like diffuse and specular reflection and is based on the MTL material format of Lightwave's OBJ scene files. To create an OBJ material pass the type string "`obj`" to `ospNewMaterial`. Its main parameters are

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | kd | white 0.8 | diffuse color |
| vec3f | ks | black | specular color |
| float | ns | 10 | shininess (Phong exponent), usually in $[2–10^4]$ |
| float | d | opaque | opacity |
| vec3f | tf | black | transparency filter color |
| OSPTexture | map_bump | NULL | normal map |

Table 3.37 – Main parameters of the OBJ material.

In particular when using the path tracer it is important to adhere to the principle of energy conservation, i.e., that the amount of light reflected by a surface is not larger than the light arriving. Therefore the path tracer issues a warning and renormalizes the color parameters if the sum of `Kd`, `Ks`, and `Tf` is larger than one in any color channel. Similarly important to mention is that almost all materials of the real world reflect at most only about 80% of the incoming light. So even for a white sheet of paper or white wall paint do better not set `Kd` larger than 0.8; otherwise rendering times are unnecessary long and the contrast in the final images is low (for example, the corners of a white room would hardly be discernible, as can be seen in the figure below).



Figure 3.7 – Comparison of diffuse rooms with 100% reflecting white paint (left) and realistic 80% reflecting white paint (right), which leads to higher overall contrast. Note that exposure has been adjusted to achieve similar brightness levels.

If present, the color component of geometries is also used for the diffuse color `Kd` and the alpha component is also used for the opacity `d`.

Normal mapping can simulate small geometric features via the texture `map_Bump`. The normals $n$ in the normal map are with respect to the local tangential shading coordinate system and are encoded as $1/2(n+1)$, thus a texel $(0.5, 0.5, 1)$[6] represents the unperturbed shading normal $(0, 0, 1)$. Because of this encoding an sRGB gamma texture format is ignored and normals are always

[6] respectively $(127, 127, 255)$ for 8 bit textures and $(32767, 32767, 65535)$ for 16 bit textures

fetched as linear from a normal map. Note that the orientation of normal maps is important for a visually consistent look: by convention OSPRay uses a coordinate system with the origin in the lower left corner; thus a convexity will look green toward the top of the texture image (see also the example image of a normal map). If this is not the case flip the normal map vertically or invert its green channel.

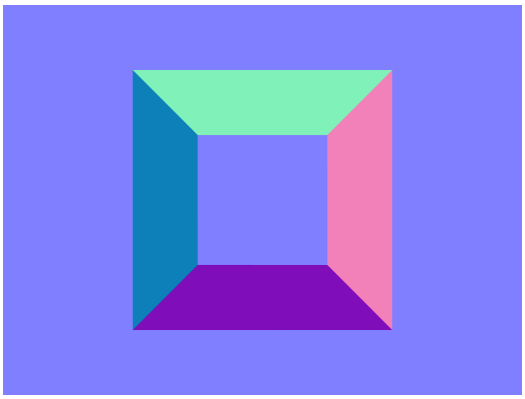

Figure 3.8 – Normal map representing an exalted square pyramidal frustum.

Note that currently only the path tracer implements colored transparency with `Tf` and normal mapping with `map_Bump`.

All parameters (except `Tf`) can be textured by passing a texture handle, prefixed with "`map_`". The fetched texels are multiplied by the respective parameter value. If only the texture is given (but not the corresponding parameter), only the texture is used (the default value of the parameter is *not* multiplied). The color textures `map_Kd` and `map_Ks` are typically in one of the sRGB gamma encoded formats, whereas textures `map_Ns` and `map_d` are usually in a linear format (and only the first component is used). Additionally, all textures support texture transformations.

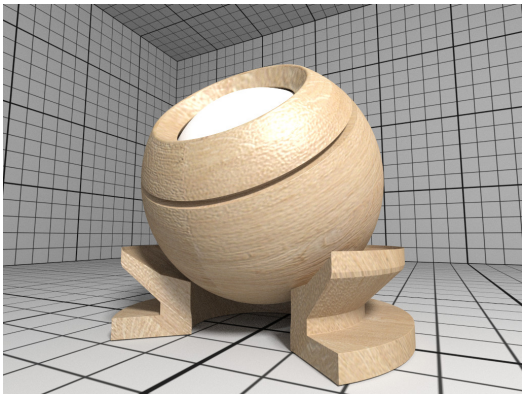

Figure 3.9 – Rendering of a OBJ material with wood textures.

#### 3.7.4.2 Principled

The Principled material is the most complex material offered by the path tracer, which is capable of producing a wide variety of materials (e.g., plastic, metal, wood, glass) by combining multiple different layers and lobes. It uses the GGX microfacet distribution with approximate multiple scattering for dielectrics and metals, uses the Oren-Nayar model for diffuse reflection, and is energy conserving. To create a Principled material, pass the type string "`principled`" to `ospNewMaterial`. Its parameters are listed in the table below.

All parameters can be textured by passing a texture handle, prefixed with "`map_`" (e.g., "`map_baseColor`"). texture transformations are supported as well.

Table 3.38 – Parameters of the Principled material.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | baseColor | white 0.8 | base reflectivity (diffuse and/or metallic) |
| vec3f | edgeColor | white | edge tint (metallic only) |
| float | metallic | 0 | mix between dielectric (diffuse and/or specular) and metallic (specular only with complex IOR) in [0–1] |
| float | diffuse | 1 | diffuse reflection weight in [0–1] |
| float | specular | 1 | specular reflection/transmission weight in [0–1] |
| float | ior | 1 | dielectric index of refraction |
| float | transmission | 0 | specular transmission weight in [0–1] |
| vec3f | transmissionColor | white | attenuated color due to transmission (Beer's law) |
| float | transmissionDepth | 1 | distance at which color attenuation is equal to transmissionColor |
| float | roughness | 0 | diffuse and specular roughness in [0–1], 0 is perfectly smooth |
| float | anisotropy | 0 | amount of specular anisotropy in [0–1] |
| float | rotation | 0 | rotation of the direction of anisotropy in [0–1], 1 is going full circle |
| float | normal | 1 | default normal map/scale for all layers |
| float | baseNormal | 1 | base normal map/scale (overrides default normal) |
| bool | thin | false | flag specifying whether the material is thin or solid |
| float | thickness | 1 | thickness of the material (thin only), affects the amount of color attenuation due to specular transmission |
| float | backlight | 0 | amount of diffuse transmission (thin only) in [0–2], 1 is 50% reflection and 50% transmission, 2 is transmission only |
| float | coat | 0 | clear coat layer weight in [0–1] |
| float | coatIor | 1.5 | clear coat index of refraction |
| vec3f | coatColor | white | clear coat color tint |
| float | coatThickness | 1 | clear coat thickness, affects the amount of color attenuation |
| float | coatRoughness | 0 | clear coat roughness in [0–1], 0 is perfectly smooth |
| float | coatNormal | 1 | clear coat normal map/scale (overrides default normal) |
| float | sheen | 0 | sheen layer weight in [0–1] |
| vec3f | sheenColor | white | sheen color tint |
| float | sheenTint | 0 | how much sheen is tinted from sheenColor toward baseColor |
| float | sheenRoughness | 0.2 | sheen roughness in [0–1], 0 is perfectly smooth |
| float | opacity | 1 | cut-out opacity/transparency, 1 is fully opaque |

#### 3.7.4.3 CarPaint

The CarPaint material is a specialized version of the Principled material for rendering different types of car paints. To create a CarPaint material, pass the type string "`carPaint`" to `ospNewMaterial`. Its parameters are listed in the table below.

All parameters can be textured by passing a texture handle, prefixed with "`map_`" (e.g., "`map_baseColor`"). texture transformations are supported as well.

#### 3.7.4.4 Metal

The path tracer offers a physical metal, supporting changing roughness and realistic color shifts at edges. To create a Metal material pass the type string "`metal`"

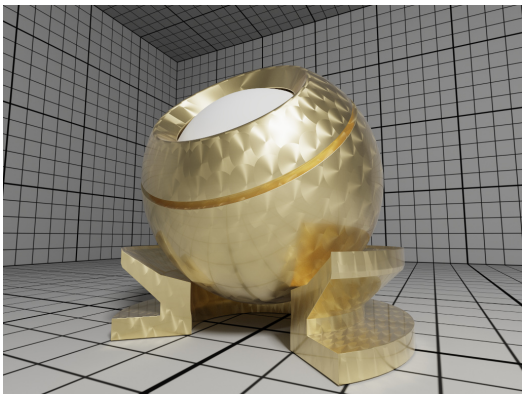

Figure 3.10 – Rendering of a Principled coated brushed metal material with textured anisotropic rotation and a dust layer (sheen) on top.

Table 3.39 – Parameters of the CarPaint material.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | baseColor | white 0.8 | diffuse base reflectivity |
| float | roughness | 0 | diffuse roughness in [0–1], 0 is perfectly smooth |
| float | normal | 1 | normal map/scale |
| vec3f float | flakeColor flakeDensity | Aluminium 0 | color of metallic flakes density of metallic flakes in [0–1], 0 disables flakes, 1 fully covers the surface with flakes |
| float | flakeScale | 100 | scale of the flake structure, higher values increase the amount of flakes |
| float | flakeSpread | 0.3 | flake spread in [0–1] |
| float | flakeJitter | 0.75 | flake randomness in [0–1] |
| float | flakeRoughness | 0.3 | flake roughness in [0–1], 0 is perfectly smooth |
| float | coat | 1 | clear coat layer weight in [0–1] |
| float | coatIor | 1.5 | clear coat index of refraction |
| vec3f | coatColor | white | clear coat color tint |
| float | coatThickness | 1 | clear coat thickness, affects the amount of color attenuation |
| float | coatRoughness | 0 | clear coat roughness in [0–1], 0 is perfectly smooth |
| float | coatNormal | 1 | clear coat normal map/scale |
| vec3f | flipflopColor | white | reflectivity of coated flakes at grazing angle, used together with coatColor produces a pearlescent paint |
| float | flipflopFalloff | 1 | flip flop color falloff, 1 disables the flip flop effect |

to `ospNewMaterial`. Its parameters are

Table 3.40 – Parameters of the Metal material.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f[] | ior | Aluminium | data array of spectral samples of complex refractive index, each entry in the form (wavelength, eta, k), ordered by wavelength (which is in nm) |
| vec3f | eta | | RGB complex refractive index, real part |
| vec3f | k | | RGB complex refractive index, imaginary part |
| float | roughness | 0.1 | roughness in [0–1], 0 is perfect mirror |

The main appearance (mostly the color) of the Metal material is controlled by the physical parameters `eta` and `k`, the wavelength-dependent, complex index

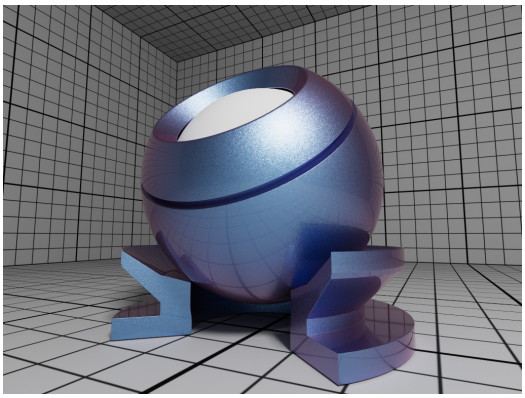

Figure 3.11 – Rendering of a pearlescent CarPaint material.

of refraction. These coefficients are quite counter-intuitive but can be found in published measurements. For accuracy the index of refraction can be given as an array of spectral samples in `ior`, each sample a triplet of wavelength (in nm), eta, and k, ordered monotonically increasing by wavelength; OSPRay will then calculate the Fresnel in the spectral domain. Alternatively, `eta` and `k` can also be specified as approximated RGB coefficients; some examples are given in below table.

| Metal | eta | k |
|---|---|---|
| Ag, Silver | (0.051, 0.043, 0.041) | (5.3, 3.6, 2.3) |
| Al, Aluminium | (1.5, 0.98, 0.6) | (7.6, 6.6, 5.4) |
| Au, Gold | (0.07, 0.37, 1.5) | (3.7, 2.3, 1.7) |
| Cr, Chromium | (3.2, 3.1, 2.3) | (3.3, 3.3, 3.1) |
| Cu, Copper | (0.1, 0.8, 1.1) | (3.5, 2.5, 2.4) |

Table 3.41 – Index of refraction of selected metals as approximated RGB coefficients, based on data from https://refractiveindex.info/.

The `roughness` parameter controls the variation of microfacets and thus how polished the metal will look. The roughness can be modified by a texture `map_roughness` (texture transformations are supported as well) to create notable edging effects.

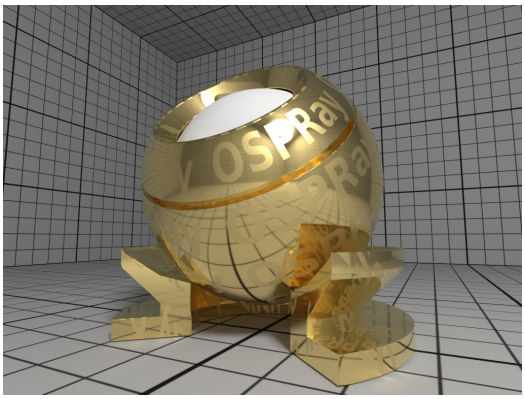

Figure 3.12 – Rendering of golden Metal material with textured roughness.

#### 3.7.4.5 Alloy

The path tracer offers an alloy material, which behaves similar to Metal, but allows for more intuitive and flexible control of the color. To create an Alloy material pass the type string "`alloy`" to `ospNewMaterial`. Its parameters are

The main appearance of the Alloy material is controlled by the parameter `color`, while `edgeColor` influences the tint of reflections when seen at grazing

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | color | white 0.9 | reflectivity at normal incidence (0 degree) |
| vec3f | edgeColor | white | reflectivity at grazing angle (90 degree) |
| float | roughness | 0.1 | roughness, in [0–1], 0 is perfect mirror |

Table 3.42 – Parameters of the Alloy material.

angles (for real metals this is always 100% white). If present, the color component of geometries is also used for reflectivity at normal incidence `color`. As in Metal the `roughness` parameter controls the variation of microfacets and thus how polished the alloy will look. All parameters can be textured by passing a texture handle, prefixed with "`map_`"; texture transformations are supported as well.

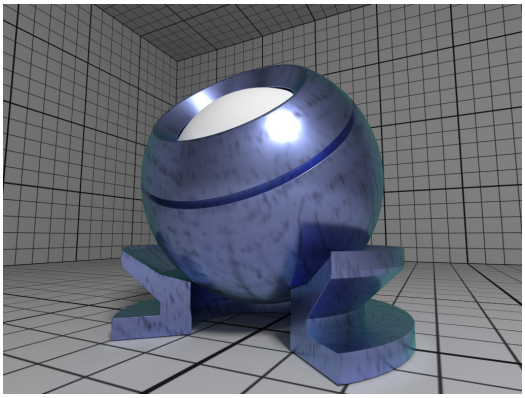

Figure 3.13 – Rendering of a fictional Alloy material with textured color.

#### 3.7.4.6 Glass

The path tracer offers a realistic a glass material, supporting refraction and volumetric attenuation (i.e., the transparency color varies with the geometric thickness). To create a Glass material pass the type string "`glass`" to `ospNewMaterial`. Its parameters are

| Type | Name | Default | Description |
|---|---|---|---|
| float | eta | 1.5 | index of refraction |
| vec3f | attenuationColor | white | resulting color due to attenuation |
| float | attenuationDistance | 1 | distance affecting attenuation |

Table 3.43 – Parameters of the Glass material.

For convenience, the rather counter-intuitive physical attenuation coefficients will be calculated from the user inputs in such a way, that the `attenuationColor` will be the result when white light traveled trough a glass of thickness `attenuationDistance`.

#### 3.7.4.7 ThinGlass

The path tracer offers a thin glass material useful for objects with just a single surface, most prominently windows. It models a thin, transparent slab, i.e., it behaves as if a second, virtual surface is parallel to the real geometric surface. The implementation accounts for multiple internal reflections between the interfaces (including attenuation), but neglects parallax effects due to its (virtual) thickness. To create a such a thin glass material pass the type string "`thinGlass`" to `ospNewMaterial`. Its parameters are

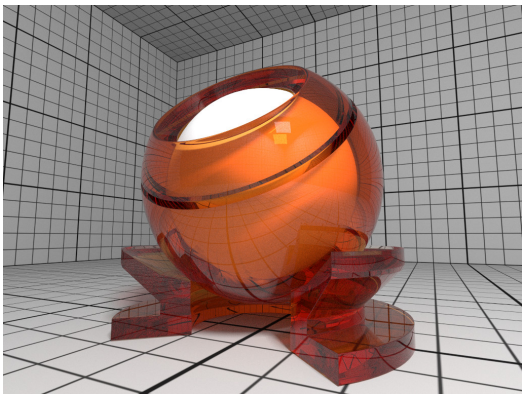

Figure 3.14 – Rendering of a Glass material with orange attenuation.

| Type | Name | Default | Description |
|---|---|---|---|
| float | eta | 1.5 | index of refraction |
| vec3f | attenuationColor | white | resulting color due to attenuation |
| float | attenuationDistance | 1 | distance affecting attenuation |
| float | thickness | 1 | virtual thickness |

Table 3.44 – Parameters of the ThinGlass material.

For convenience the attenuation is controlled the same way as with the Glass material. Additionally, the color due to attenuation can be modulated with a texture `map_attenuationColor` (texture transformations are supported as well). If present, the color component of geometries is also used for the attenuation color. The `thickness` parameter sets the (virtual) thickness and allows for easy exchange of parameters with the (real) Glass material; internally just the ratio between `attenuationDistance` and `thickness` is used to calculate the resulting attenuation and thus the material appearance.

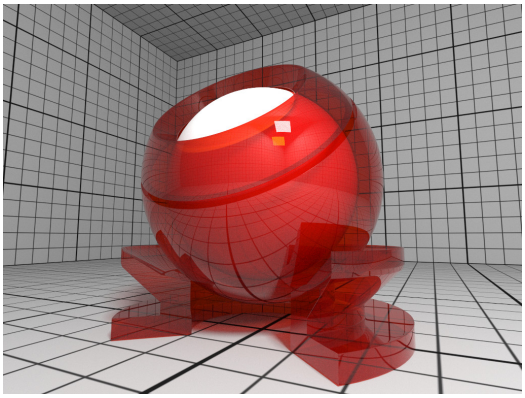

Figure 3.15 – Rendering of a ThinGlass material with red attenuation.

#### 3.7.4.8 MetallicPaint

The path tracer offers a metallic paint material, consisting of a base coat with optional flakes and a clear coat. To create a MetallicPaint material pass the type string "`metallicPaint`" to `ospNewMaterial`. Its parameters are listed in the table below.

The color of the base coat `baseColor` can be textured by a texture `map_baseColor`, which also supports texture transformations. If present, the color component of geometries is also used for the color of the base coat. Parameter `flakeAmount` controls the proportion of flakes in the base coat, so when setting it to 1 the `baseColor` will not be visible. The shininess of the metallic component

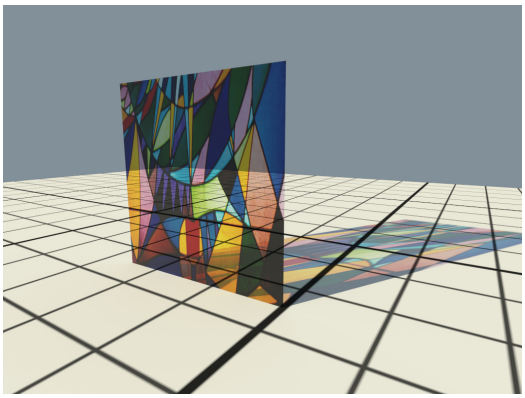

Figure 3.16 – Example image of a colored window made with textured attenuation of the ThinGlass material.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | baseColor | white 0.8 | color of base coat |
| float | flakeAmount | 0.3 | amount of flakes, in [0–1] |
| vec3f | flakeColor | Aluminium | color of metallic flakes |
| float | flakeSpread | 0.5 | spread of flakes, in [0–1] |
| float | eta | 1.5 | index of refraction of clear coat |

Table 3.45 – Parameters of the MetallicPaint material.

is governed by `flakeSpread`, which controls the variation of the orientation of the flakes, similar to the `roughness` parameter of Metal. Note that the effect of the metallic flakes is currently only computed on average, thus individual flakes are not visible.

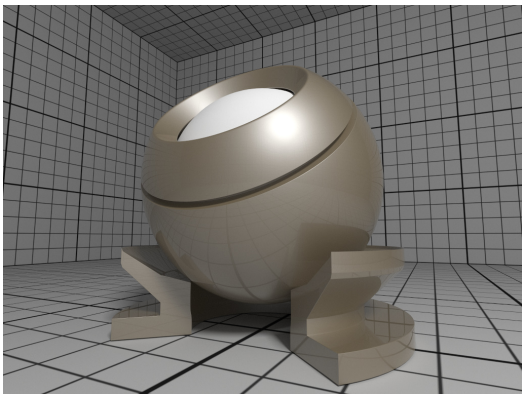

Figure 3.17 – Rendering of a MetallicPaint material.

#### 3.7.4.9 Luminous

The path tracer supports the Luminous material which emits light uniformly in all directions and which can thus be used to turn any geometric object into a light source[7]. It is created by passing the type string "`luminous`" to `ospNewMaterial`. The amount of constant radiance that is emitted is determined by combining the general parameters of lights: `color` and `intensity`.

[7] If `geometryLights` is enabled in the path tracer.

| Type | Name | Default | Description |
|---|---|---|---|
| vec3f | color | white | color of the emitted light |
| float | intensity | 1 | intensity of the light (a factor) |
| float | transparency | 1 | material transparency |

Table 3.46 – Parameters accepted by the Luminous material.

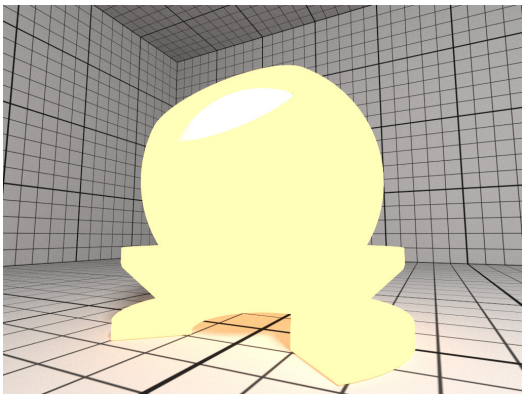

Figure 3.18 – Rendering of a yellow Luminous material.

### 3.7.5 Texture

OSPRay currently implements two texture types (`texture2d` and `volume`) and is open for extension to other types by applications. More types may be added in future releases.

To create a new texture use

```
OSPTexture ospNewTexture(const char *type);
```

#### 3.7.5.1 Texture2D

The `texture2d` texture type implements an image-based texture, where its parameters are as follows

| Type | Name | Description |
|---|---|---|
| int | format | `OSPTextureFormat` for the texture |
| int | filter | default `OSP_TEXTURE_FILTER_BILINEAR`, alternatively `OSP_TEXTURE_FILTER_NEAREST` |
| OSPData | data | the actual texel 2D data |

Table 3.47 – Parameters of `texture2d` texture type.

The supported texture formats for `texture2d` are:

The size of the texture is inferred from the size of the 2D array `data`, which also needs have a compatible type to `format`. The texel data in `data` starts with the texels in the lower left corner of the texture image, like in OpenGL. Per default a texture fetch is filtered by performing bi-linear interpolation of the nearest 2×2 texels; if instead fetching only the nearest texel is desired (i.e., no filtering) then pass the `OSP_TEXTURE_FILTER_NEAREST` flag.

Texturing with `texture2d` image textures requires geometries with texture coordinates, e.g., a mesh with `vertex.texcoord` provided.

#### 3.7.5.2 Volume Texture

The `volume` texture type implements texture lookups based on 3D object coordinates of the surface hit point on the associated geometry. If the given hit point is within the attached volume, the volume is sampled and classified with the transfer function attached to the volume. This implements the ability to visualize volume values (as colored by a transfer function) on arbitrary surfaces inside the volume (as opposed to an isosurface showing a particular value in the volume). Its parameters are as follows

TextureVolume can be used for implementing slicing of volumes with any geometry type. It enables coloring of the slicing geometry with a different transfer function than that of the sliced volume.

| Name | Description |
|---|---|
| OSP_TEXTURE_RGBA8 | 8 bit [0–255] linear components red, green, blue, alpha |
| OSP_TEXTURE_SRGBA | 8 bit sRGB gamma encoded color components, and linear alpha |
| OSP_TEXTURE_RGBA32F | 32 bit float components red, green, blue, alpha |
| OSP_TEXTURE_RGB8 | 8 bit [0–255] linear components red, green, blue |
| OSP_TEXTURE_SRGB | 8 bit sRGB gamma encoded components red, green, blue |
| OSP_TEXTURE_RGB32F | 32 bit float components red, green, blue |
| OSP_TEXTURE_R8 | 8 bit [0–255] linear single component red |
| OSP_TEXTURE_RA8 | 8 bit [0–255] linear two components red, alpha |
| OSP_TEXTURE_L8 | 8 bit [0–255] gamma encoded luminance (replicated into red, green, blue) |
| OSP_TEXTURE_LA8 | 8 bit [0–255] gamma encoded luminance, and linear alpha |
| OSP_TEXTURE_R32F | 32 bit float single component red |
| OSP_TEXTURE_RGBA16 | 16 bit [0–65535] linear components red, green, blue, alpha |
| OSP_TEXTURE_RGB16 | 16 bit [0–65535] linear components red, green, blue |
| OSP_TEXTURE_RA16 | 16 bit [0–65535] linear two components red, alpha |
| OSP_TEXTURE_R16 | 16 bit [0–65535] linear single component red |

Table 3.48 – Supported texture formats by `texture2d`, i.e., valid constants of type `OSPTextureFormat`.

| Type | Name | Description |
|---|---|---|
| OSPVolume | volume | Volume used to generate color lookups |
| OSPTransferFunction | transferFunction | [TransferFunction] applied to `volume` |

Table 3.49 – Parameters of `volume` texture type.

#### 3.7.5.3 Texture Transformations

All materials with textures also offer to manipulate the placement of these textures with the help of texture transformations. If so, this convention shall be used: the following parameters are prefixed with "`texture_name.`").

| Type | Name | Description |
|---|---|---|
| linear2f | transform | linear transformation (rotation, scale) |
| float | rotation | angle in degree, counterclockwise, around center |
| vec2f | scale | enlarge texture, relative to center (0.5, 0.5) |
| vec2f | translation | move texture in positive direction (right/up) |

Table 3.50 – Parameters to define 2D texture coordinate transformations.

Above parameters are combined into a single `affine2d` transformation matrix and the transformations are applied in the given order. Rotation, scale and translation are interpreted "texture centric", i.e., their effect seen by an user are

relative to the texture (although the transformations are applied to the texture coordinates).

| Type | Name | Description |
|---|---|---|
| affine3f | transform | linear transformation (rotation, scale) plus translation |

Table 3.51 – Parameter to define 3D volume texture transformations.

Similarly, volume texture placement can also be modified by an `affine3f` transformation matrix.

### 3.7.6 Cameras

To create a new camera of given type `type` use

```
OSPCamera ospNewCamera(const char *type);
```

All cameras accept these parameters:

| Type | Name | Description |
|---|---|---|
| vec3f | position | position of the camera in world-space |
| vec3f | direction | main viewing direction of the camera |
| vec3f | up | up direction of the camera |
| float | nearClip | near clipping distance |
| vec2f | imageStart | start of image region (lower left corner) |
| vec2f | imageEnd | end of image region (upper right corner) |

Table 3.52 – Parameters accepted by all cameras.

The camera is placed and oriented in the world with `position`, `direction` and `up`. OSPRay uses a right-handed coordinate system. The region of the camera sensor that is rendered to the image can be specified in normalized screen-space coordinates with `imageStart` (lower left corner) and `imageEnd` (upper right corner). This can be used, for example, to crop the image, to achieve asymmetrical view frusta, or to horizontally flip the image to view scenes which are specified in a left-handed coordinate system. Note that values outside the default range of $[0–1]$ are valid, which is useful to easily realize overscan or film gate, or to emulate a shifted sensor.

#### 3.7.6.1 Perspective Camera

The perspective camera implements a simple thin lens camera for perspective rendering, supporting optionally depth of field and stereo rendering, but no motion blur. It is created by passing the type string "`perspective`" to `ospNewCamera`. In addition to the general parameters understood by all cameras the perspective camera supports the special parameters listed in the table below.

Note that when computing the `aspect` ratio a potentially set image region (using `imageStart` & `imageEnd`) needs to be regarded as well.

In architectural photography it is often desired for aesthetic reasons to display the vertical edges of buildings or walls vertically in the image as well, regardless of how the camera is tilted. Enabling the `architectural` mode achieves this by internally leveling the camera parallel to the ground (based on the `up` direction) and then shifting the lens such that the objects in direction `dir` are centered in the image. If finer control of the lens shift is needed use `imageStart` & `imageEnd`. Because the camera is now effectively leveled its image plane and thus the plane of focus is oriented parallel to the front of buildings, the whole façade

Table 3.53 – Additional parameters accepted by the perspective camera.

| Type | Name | Description |
|---|---|---|
| float | fovy | the field of view (angle in degree) of the frame's height |
| float | aspect | ratio of width by height of the frame (and image region) |
| float | apertureRadius | size of the aperture, controls the depth of field |
| float | focusDistance | distance at where the image is sharpest when depth of field is enabled |
| bool | architectural | vertical edges are projected to be parallel |
| uchar | stereoMode | `OSPStereoMode` for stereo rendering, possible values are: |
| | | `OSP_STEREO_NONE` (default) |
| | | `OSP_STEREO_LEFT` |
| | | `OSP_STEREO_RIGHT` |
| | | `OSP_STEREO_SIDE_BY_SIDE` |
| | | `OSP_STEREO_TOP_BOTTOM` (left eye at top half) |
| float | interpupillaryDistance | distance between left and right eye when stereo is enabled, default 0.0635 |

appears sharp, as can be seen in the example images below. The resolution of the framebuffer is not altered by `imageStart`/`imageEnd`.

#### 3.7.6.2 Orthographic Camera

The orthographic camera implements a simple camera with orthographic projection, without support for depth of field or motion blur. It is created by passing the type string "`orthographic`" to `ospNewCamera`. In addition to the general parameters understood by all cameras the orthographic camera supports the following special parameters:

Table 3.54 – Additional parameters accepted by the orthographic camera.

| Type | Name | Description |
|---|---|---|
| float | height | size of the camera's image plane in y, in world coordinates |
| float | aspect | ratio of width by height of the frame |

For convenience the size of the camera sensor, and thus the extent of the scene that is captured in the image, can be controlled with the `height` parameter. The same effect can be achieved with `imageStart` and `imageEnd`, and both methods can be combined. In any case, the `aspect` ratio needs to be set accordingly to get an undistorted image.

#### 3.7.6.3 Panoramic Camera

The panoramic camera implements a simple camera with support for stereo rendering. It captures the complete surrounding with a latitude / longitude mapping and thus the rendered images should best have a ratio of 2:1. A panoramic camera is created by passing the type string "`panoramic`" to `ospNewCamera`. It is placed and oriented in the scene by using the general parameters understood by all cameras.

### 3.7.7 Picking

To get the world-space position of the geometry (if any) seen at [0–1] normalized screen-space pixel coordinates `screenPos` use

Figure 3.19 – Example image created with the perspective camera, featuring depth of field.

Figure 3.20 – Enabling the `architectural` flag corrects the perspective projection distortion, resulting in parallel vertical edges.

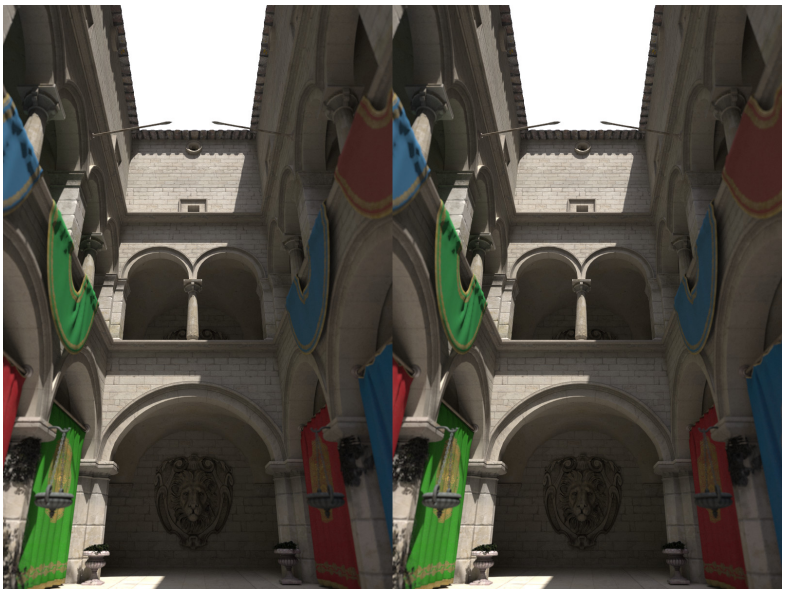

Figure 3.21 – Example 3D stereo image using `stereoMode = OSP_STEREO_SIDE_BY_SIDE`.

Figure 3.22 – Example image created with the orthographic camera.

Figure 3.23 – Latitude / longitude map created with the panoramic camera.

Table 3.55 – Additional parameters accepted by the panoramic camera.

| Type | Name | Description |
|---|---|---|
| uchar | stereoMode | `OSPStereoMode` for stereo rendering, possible values are: |
| | | `OSP_STEREO_NONE` (default) |
| | | `OSP_STEREO_LEFT` |
| | | `OSP_STEREO_RIGHT` |
| | | `OSP_STEREO_SIDE_BY_SIDE` |
| | | `OSP_STEREO_TOP_BOTTOM` (left eye at top half) |
| float | interpupillaryDistance | distance between left and right eye when stereo is enabled, default 0.0635 |

```
void ospPick(OSPPickResult *,
    OSPFrameBuffer,
    OSPRenderer,
    OSPCamera,
    OSPWorld,
    osp_vec2f screenPos);
```

The result is returned in the provided `OSPPickResult` struct:

```
typedef struct {
    int hasHit;
    osp_vec3f worldPosition;
    OSPGeometricModel GeometricModel;
    uint32_t primID;
} OSPPickResult;
```

Note that `ospPick` considers exactly the same camera of the given renderer that is used to render an image, thus matching results can be expected. If the camera supports depth of field then the center of the lens and thus the center of the circle of confusion is used for picking.

## 3.8 Framebuffer

The framebuffer holds the rendered 2D image (and optionally auxiliary information associated with pixels). To create a new framebuffer object of given size `size` (in pixels), color format, and channels use

```
OSPFrameBuffer ospNewFrameBuffer(int size_x, int size_y,
    OSPFrameBufferFormat format = OSP_FB_SRGBA,
    uint32_t frameBufferChannels = OSP_FB_COLOR);
```

The parameter `format` describes the format the color buffer has *on the host*, and the format that `ospMapFrameBuffer` will eventually return. Valid values are:

The parameter `frameBufferChannels` specifies which channels the framebuffer holds, and can be combined together by bitwise OR from the values of `OSPFrameBufferChannel` listed in the table below.

If a certain channel value is *not* specified, the given buffer channel will not be present. Note that OSPRay makes a clear distinction between the *external* format of the framebuffer and the internal one: The external format is the format the user specifies in the `format` parameter; it specifies what color format OSPRay will eventually *return* the framebuffer to the application (when calling `ospMapFrameBuffer`): no matter what OSPRay uses internally, it will simply

| Name | Description |
|---|---|
| OSP_FB_NONE | framebuffer will not be mapped by the application |
| OSP_FB_RGBA8 | 8 bit [0–255] linear component red, green, blue, alpha |
| OSP_FB_SRGBA | 8 bit sRGB gamma encoded color components, and linear alpha |
| OSP_FB_RGBA32F | 32 bit float components red, green, blue, alpha |

Table 3.56 – Supported color formats of the framebuffer that can be passed to `ospNewFrameBuffer`, i.e., valid constants of type `OSPFrameBufferFormat`.

| Name | Description |
|---|---|
| OSP_FB_COLOR | RGB color including alpha |
| OSP_FB_DEPTH | euclidean distance to the camera (*not* to the image plane), as linear 32 bit float; for multiple samples per pixel their minimum is taken |
| OSP_FB_ACCUM | accumulation buffer for progressive refinement |
| OSP_FB_VARIANCE | for estimation of the current noise level if OSP_FB_ACCUM is also present, see rendering |
| OSP_FB_NORMAL | accumulated world-space normal of the first hit, as vec3f |
| OSP_FB_ALBEDO | accumulated material albedo (color without illumination) at the first hit, as vec3f |

Table 3.57 – Framebuffer channels constants (of type `OSPFrameBufferChannel`), naming optional information the framebuffer can store. These values can be combined by bitwise OR when passed to `ospNewFrameBuffer`.

return a 2D array of pixels of that format, with possibly all kinds of reformatting, compression/decompression, etc., going on in-between the generation of the *internal* framebuffer and the mapping of the externally visible one.

In particular, `OSP_FB_NONE` is a perfectly valid pixel format for a framebuffer that an application will never map. For example, an application driving a display wall may well generate an intermediate framebuffer and eventually transfer its pixel to the individual displays using an `OSPImageOperation` image operation.

The application can map the given channel of a framebuffer – and thus access the stored pixel information – via

```
const void *ospMapFrameBuffer(OSPFrameBuffer, OSPFrameBufferChannel = OSP_FB_COLOR);
```

Note that `OSP_FB_ACCUM` or `OSP_FB_VARIANCE` cannot be mapped. The origin of the screen coordinate system in OSPRay is the lower left corner (as in OpenGL), thus the first pixel addressed by the returned pointer is the lower left pixel of the image.

A previously mapped channel of a framebuffer can be unmapped by passing the received pointer `mapped` to

```
void ospUnmapFrameBuffer(const void *mapped, OSPFrameBuffer);
```

The individual channels of a framebuffer can be cleared with

```
void ospResetAccumulation(OSPFrameBuffer);
```

This function will clear *all* accumulating buffers (`OSP_FB_VARIANCE`, `OSP_FB_NORMAL`, and `OSP_FB_ALBEDO`, if present) and resets the accumulation counter `accumID`. It is unspecified if the existing color and depth buffers are physically cleared when `ospResetAccumulation` is called.

If `OSP_FB_VARIANCE` is specified, an estimate of the variance of the last accumulated frame can be queried with

```
float ospGetVariance(OSPFrameBuffer);
```

Note this value is only updated after synchronizing with `OSP_FRAME_FINISHED`, as further described in asynchronous rendering. The estimated variance can be used by the application as a quality indicator and thus to decide whether to stop or to continue progressive rendering.

The framebuffer takes a list of pixel operations to be applied to the image in sequence as an `OSPData`. The pixel operations will be run in the order they are in the array.

| Type | Name | Description |
|---|---|---|
| OSPImageOperation[] | imageOperation | ordered sequence of image operations |

Table 3.58 – Parameters accepted by the framebuffer.

### 3.8.1 Image Operation

Image operations are functions that are applied to every pixel of a frame. Examples include post-processing, filtering, blending, tone mapping, or sending tiles to a display wall. To create a new pixel operation of given type `type` use

```
OSPImageOperation ospNewImageOperation(const char *type);
```

#### 3.8.1.1 Tone Mapper

The tone mapper is a pixel operation which implements a generic filmic tone mapping operator. Using the default parameters it approximates the Academy Color Encoding System (ACES). The tone mapper is created by passing the type string "`tonemapper`" to `ospNewImageOperation`. The tone mapping curve can be customized using the parameters listed in the table below.

Table 3.59 – Parameters accepted by the tone mapper.

| Type | Name | Default | Description |
|---|---|---|---|
| float | exposure | 1.0 | amount of light per unit area |
| float | contrast | 1.6773 | contrast (toe of the curve); typically is in [1–2] |
| float | shoulder | 0.9714 | highlight compression (shoulder of the curve); typically is in [0.9–1] |
| float | midIn | 0.18 | mid-level anchor input; default is 18% gray |
| float | midOut | 0.18 | mid-level anchor output; default is 18% gray |
| float | hdrMax | 11.0785 | maximum HDR input that is not clipped |
| bool | acesColor | true | apply the ACES color transforms |

To use the popular "Uncharted 2" filmic tone mapping curve instead, set the parameters to the values listed in the table below.

| Name | Value |
|---|---|
| contrast | 1.1759 |
| shoulder | 0.9746 |
| midIn | 0.18 |
| midOut | 0.18 |
| hdrMax | 6.3704 |
| acesColor | false |

Table 3.60 – Filmic tone mapping curve parameters. Note that the curve includes an exposure bias to match 18% middle gray.

#### 3.8.1.2 Denoiser

OSPRay comes with a module that adds support for Intel® Open Image Denoise. This is provided as an optional module as it creates an additional project dependency at compile time. The module implements a "`denoiser`" frame operation, which denoises the entire frame before the frame is completed.

## 3.9 Rendering

### 3.9.1 Asynchronous Rendering

Rendering is by default asynchronous (non-blocking), and is done by combining a framebuffer, renderer, camera, and world.

What to render and how to render it depends on the renderer's parameters. If the framebuffer supports accumulation (i.e., it was created with `OSP_FB_ACCUM`) then successive calls to `ospRenderFrame` will progressively refine the rendered image.

To start an render task, use

```
OSPFuture ospRenderFrame(OSPFrameBuffer, OSPRenderer, OSPCamera, OSPWorld);
```

This returns an `OSPFuture` handle, which can be used to synchronize with the application, cancel, or query for progress of the running task. When `ospRenderFrame` is called, there is no guarantee when the associated task will begin execution.

Progress of a running frame can be queried with the following API function

```
float ospGetProgress(OSPFuture);
```

This returns the progress of the task in [0-1].

Applications can wait on the result of an asynchronous operation, or choose to only synchronize with a specific event. To synchronize with an `OSPFuture` use

```
void ospWait(OSPFuture, OSPSyncEvent = OSP_TASK_FINISHED);
```

The following are values which can be synchronized with the application

Table 3.61 – Supported events that can be passed to `ospWait`.

| Name | Description |
|---|---|
| OSP_NONE_FINISHED | Do not wait for anything to be finished (immediately return from `ospWait`) |
| OSP_WORLD_COMMITTED | Wait for the world to be committed (not yet implemented) |
| OSP_WORLD_RENDERED | Wait for the world to be rendered, but not post-processing operations (Pixel/Tile/Frame Op) |
| OSP_FRAME_FINISHED | Wait for all rendering operations to complete |
| OSP_TASK_FINISHED | Wait on full completion of the task associated with the future. The underlying task may involve one or more of the above synchronization events |

Currently only rendering can be invoked asynchronously. However, future releases of OSPRay may add more asynchronous versions of API calls (and thus return `OSPFuture`).

Applications can query whether particular events are complete with

```
int ospIsReady(OSPFuture, OSPSyncEvent = OSP_TASK_FINISHED);
```

As the given running task runs (as tracked by the `OSPFuture`), applications can query a boolean [0,1] result if the passed event has been completed.

Applications can query how long an async task ran with

```
float ospGetTaskDuration(OSPFuture);
```

This returns the wall clock execution time of the task in seconds. If the task is still running, this will block until the task is completed. This is useful for applications to query exactly how long an asynchronous task executed without the overhead of measuring both task execution + synchronization by the calling application.

### 3.9.2 Asynchronously Rendering and ospCommit()

The use of either `ospRenderFrame` or `ospRenderFrame` requires that all objects in the scene being rendered have been committed before rendering occurs. If a call to `ospCommit()` happens while a frame is rendered, the result is undefined behavior and should be avoided.

### 3.9.3 Synchronous Rendering

For convenience in certain use cases, `ospray_util.h` provides a synchronous version of `ospRenderFrame`:

```
float ospRenderFrameBlocking(OSPFrameBuffer, OSPRenderer, OSPCamera, OSPWorld);
```

This version is the equivalent of:

```
ospRenderFrame
ospWait(f, OSP_TASK_FINISHED)
return ospGetVariance(fb)
```

This version is closest to `ospRenderFrame` from OSPRay v1.x.

## 3.10 Distributed rendering with MPI

The OSPRay MPI module is now a stand alone repository. It can be found on GitHub here, where all code and documentation can be found.

# Chapter 4
# Examples

## 4.1 Tutorial

A minimal working example demonstrating how to use OSPRay can be found at `apps/tutorials/ospTutorial.c`[1].

An example of building `ospTutorial.c` with CMake can be found in `apps/tutorials/ospTutorialFindospray/`.

To build the tutorial on Linux, build it in a build directory with

```
gcc -std=c99 ../apps/ospTutorial/ospTutorial.c \
-I ../ospray/include -L . -lospray -Wl,-rpath,. -o ospTutorial
```

On Windows build it can be build manually in a "build_directory\$Configuration" directory with

```
cl ..\..\apps\ospTutorial\ospTutorial.c -I ..\..\ospray\include -I ..\.. ospray.lib
```

Running `ospTutorial` will create two images of two triangles, rendered with the Scientific Visualization renderer with full Ambient Occlusion. The first image `firstFrame.ppm` shows the result after one call to `ospRenderFrame` – jagged edges and noise in the shadow can be seen. Calling `ospRenderFrame` multiple times enables progressive refinement, resulting in antialiased edges and converged shadows, shown after ten frames in the second image `accumulatedFrames.ppm`.

[1] A C++ version that uses the C++ convenience wrappers of OSPRay's C99 API via `include/ospray/ospray_cpp.h` is available at `apps/tutorials/ospTutorial.cpp`.

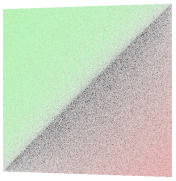

Figure 4.1 – First frame.

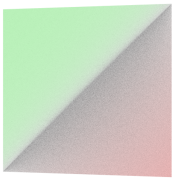

Figure 4.2 – After accumulating ten frames.

## 4.2 ospExamples

Apart from tutorials, `OSPRay` comes with a C++ app called `ospExamples` which is an elaborate easy-to-use tutorial, with a single interface to try various `OSPRay` features. It is aimed at providing users with multiple simple scenes composed of basic geometry types, lights, volumes etc. to get started with OSPRay quickly.

`ospExamples` app runs a `GLFWOSPRayWindow` instance that manages instances of the camera, framebuffer, renderer and other OSPRay objects necessary to render an interactive scene. The scene is rendered on a `GLFW` window with an `imgui` GUI controls panel for the user to manipulate the scene at runtime.

The application is located in `apps/ospExamples/` directory and can be built with CMake. It can be run from the build directory via:

```
./ospExamples <command-line-parameter>
```

The command line parameter is optional however.

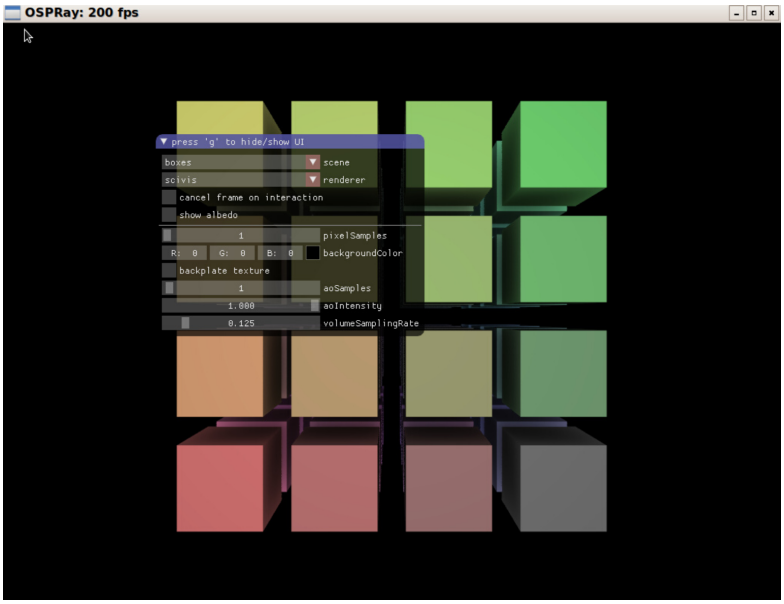


Figure 4.3 – `ospExamples` application with default `boxes` scene.

### 4.2.1 Scenes

Different scenes can be selected from the `scenes` dropdown and each scene corresponds to an instance of a special `detail::Builder` struct. Example builders are located in `apps/common/ospray_testing/builders/`. These builders provide a usage guide for the OSPRay scene hierarchy and OSPRay API in the form of `cpp` wrappers. They instantiate and manage objects for the specific scene like `cpp::Geometry`, `cpp::Volume`, `cpp::Light` etc.

The `detail::Builder` base struct is mostly responsible for setting up OSPRay `world` and objects common in all scenes (for example lighting and ground plane), which can be conveniently overridden in the derived builders.

Given below are different scenes listed with their string identifiers:

**boxes** A simple scene with `box` geometry type.

**cornell_box** A scene depicting a classic cornell box with `quad` mesh geometry type for rendering two cubes and a `quad` light type.

**curves** A simple scene with `curve` geometry type and options to change `curveBasis`. For details on different basis' please check documentation of curves.

**gravity_spheres_volume** A scene with `structuredRegular` type of volume.

**gravity_spheres_isosurface** A scene depicting iso-surface rendering of `gravity_spheres_volume` using geometry type `isosurface`.

**perlin_noise_volumes** An example scene with `structuredRegular` volume type depicting perlin noise.

**random_spheres** A simple scene depicting `sphere` geometry type.

**streamlines** A scene showcasing streamlines geometry derived from `curve` geometry type.

**subdivision_cube** A scene with a cube of `subdivision` geometry type to showcase subdivision surfaces.

**unstructured_volume** A simple scene with a volume of `unstructured` volume type.

### 4.2.2 Renderer

This app comes with three renderer options: `scivis`, `pathtracer` and `debug`. The app provides some common rendering controls like `pixelSamples` and other more specific to the renderer type like `aoIntensity` for `scivis` renderer.

The sun-sky lighting can be used in a sample scene by enabling the `renderSunSky` option of the `pathtracer` renderer. It allows the user to change `turbidity` and `sunDirection`.

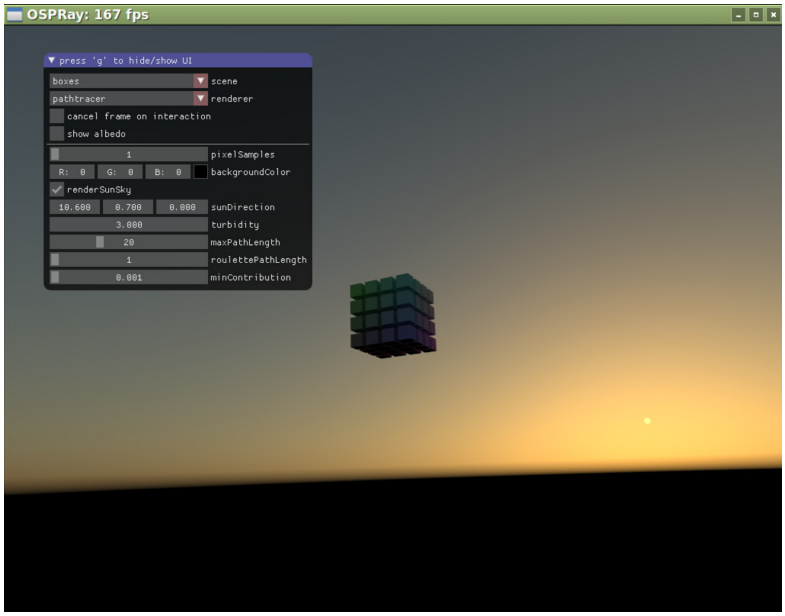


Figure 4.4 – Rendering an evening sky with the `renderSunSky` option.