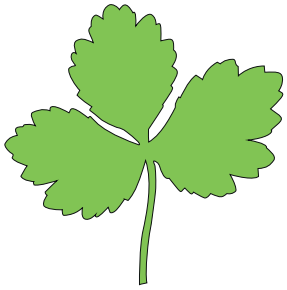


Mitsuba Documentation

Version 0.5.0

Wenzel Jakob

February 25, 2014

Contents

Part I.
Using Mitsuba

Disclaimer: This manual documents the usage, file format, and internal design of the Mitsuba rendering system. It is currently a work in progress, hence some parts may still be incomplete or missing.

1. About Mitsuba

Mitsuba is a research-oriented rendering system in the style of PBRT (`www.pbrt.org`), from which it derives much inspiration. It is written in portable C++, implements unbiased as well as biased techniques, and contains heavy optimizations targeted towards current CPU architectures. Mitsuba is extremely modular: it consists of a small set of core libraries and over 100 different plugins that implement functionality ranging from materials and light sources to complete rendering algorithms.

In comparison to other open source renderers, Mitsuba places a strong emphasis on experimental rendering techniques, such as path-based formulations of Metropolis Light Transport and volumetric modeling approaches. Thus, it may be of genuine interest to those who would like to experiment with such techniques that haven't yet found their way into mainstream renderers, and it also provides a solid foundation for research in this domain.

Other design considerations are:

Performance: Mitsuba provides optimized implementations of the most commonly used rendering algorithms. By virtue of running on a shared foundation, comparisons between them can better highlight the merits and limitations of different approaches. This is in contrast to, say, comparing two completely different rendering products, where technical information on the underlying implementation is often intentionally not provided.

Robustness: In many cases, physically-based rendering packages force the user to model scenes with the underlying algorithm (specifically: its convergence behavior) in mind. For instance, glass windows are routinely replaced with light portals, photons must be manually guided to the relevant parts of a scene, and interactions with complex materials are taboo, since they cannot be importance sampled exactly. One focus of Mitsuba will be to develop path-space light transport algorithms, which handle such cases more gracefully.

Scalability: Mitsuba instances can be merged into large clusters, which transparently distribute and jointly execute tasks assigned to them using only node-to-node communcation. It has successfully scaled to large-scale renderings that involved more than 1000 cores working on a single image. Most algorithms in Mitsuba are written using a generic parallelization layer, which can tap into this cluster-wide parallelism. The principle is that if any component of the renderer produces work that takes longer than a second or so, it at least ought to use all of the processing power it can get.

The renderer also tries to be very conservative in its use of memory, which allows it to handle large scenes (>30 million triangles) and multi-gigabyte heterogeneous volumes on consumer hardware.

Realism and accuracy: Mitsuba comes with a large repository of physically-based reflectance models for surfaces and participating media. These implementations are designed so that they can be used to build complex shader networks, while providing enough flexibility to be compatible with

a wide range of different rendering techniques, including path tracing, photon mapping, hardware-accelerated rendering and bidirectional methods.

The unbiased path tracers in Mitsuba are battle-proven and produce reference-quality results that can be used for predictive rendering, and to verify implementations of other rendering methods.

Usability: Mitsuba comes with a graphical user interface to interactively explore scenes. Once a suitable viewpoint has been found, it is straightforward to perform renderings using any of the implemented rendering techniques, while tweaking their parameters to find the most suitable settings. Experimental integration into Blender 2.5 is also available.

2. Limitations

Mitsuba can be used to solve many interesting light transport problems. However, there are some inherent limitations of the system that users should be aware of:

(i) **Wave Optics**: Mitsuba is fundamentally based on the geometric optics toolbox, which means that it generally does not simulate phenomena that arise due to the wave properties of light (diffraction, for instance).

(ii) **Polarization**: Mitsuba does not account for polarization. In other words, light is always assumed to be randomly polarized. This can be a problem for some predictive rendering applications.

(iii) **Numerical accuracy**: The accuracy of any result produced with this system is constrained by the underlying floating point computations.

For instance, an intricate scene that can be rendered without problems, may produce the wrong answer when all objects are translated away from the origin by a large distance, since floating point numbers are spaced less densely at the new position. To avoid these sorts of pitfalls, it is good to have a basic understanding of the IEEE-754 standard.

3. License

Mitsuba is free software and can be redistributed and modified under the terms of the GNU General Public License (Version 3) as provided by the Free Software Foundation.

Remarks:

- Being a "viral" license, the GPL automatically applies to all derivative work. Amongst other things, this means that without express permission, Mitsuba's source code is *off-limits* to companies that develop rendering software not distributed under a compatible license.

4. Compiling the renderer

To compile Mitsuba, you will need a recent C++ compiler (e.g. GCC 4.2+ or Visual Studio 2010/2013) and some additional libraries, which Mitsuba uses internally. Builds on all supported platforms are done using a unified system based on SCons (`http://www.scons.org`), which is a Python-based software construction tool. The exact process is different depending on which operating system is used and will be explained in the following subsections.

4.1. Common steps

To get started, you will need to download a recent version of the Mitsuba source code. Before doing this, ensure that you have read the licensing agreement (Section 15), and that you abide by its contents. Note that, being a "viral" license, the GPL automatically applies to derivative work. Amongst other things, this means that Mitsuba's source code is *off-limits* to those who develop rendering software not distributed under a compatible license.

Check that the Mercurial (`http://mercurial.selenic.com/`) versioning system[1] is installed, which is required to fetch the most recent source code release. Begin by entering the following at the command prompt (or run an equivalent command from a graphical Mercurial frontend):

```
$ hg clone https://www.mitsuba-renderer.org/hg/mitsuba
```

This should dowload a full copy of the main repository.

4.1.1. Build configurations

Common to all platforms is that a build configuration file must be selected. Several options are available on each operating system:

Linux: On Linux, there are two supported configurations:

`build/config-linux-gcc.py`: Optimized single precision GCC build. The resulting binaries include debug symbols for convenience, but these can only be used for relatively high-level debugging due to the enabled optimizations.

`build/config-linux-gcc-debug.py`: Non-optimized single precision GCC build with debug symbols. When compiled with this configuration, Mitsuba will run extremely slowly. Its main use is to track down elusive bugs.

Windows: On Windows, builds can either be performed using the Visual Studio 2010 or 2013[2] compiler or Intel XE Composer. If you are using Visual Studio 2010, note that Service Pack 1 *must* be installed or the resulting binaries will crash.

`build/config-{win32, win64}-{msvc2010, msvc2010-debug}.py`: Create 32 or 64 bit binaries using Microsoft Visual C++ version 2010. The configurations with the suffix `-debug` will include debug symbols in all binaries, which run very slowly.

[1] On Windows, you might want to use the convenient TortoiseHG shell extension (`http://tortoisehg.bitbucket.org/`) to run the subsequent steps directly from the Explorer.

[2] No other Visual Studio versions are currently supported.

build/config-win64-{msvc2013, msvc2013-debug}.py: Create 64 bit binaries using Microsoft Visual C++ version 2013. Please use Visual Studio 2010 for legacy 32 bit builds.

build/config-{win32, win64}-icl.py: Create 32 or 64 bit release binaries using Intel XE Composer (on top of Visual Studio 2010). Versions XE 2012 and 2013 are known to work.

Mac OS: On Mac OS, builds can either be performed using the the XCode 4 `llvm-gcc` toolchain or Intel XE Composer. It is possible to target MacOS 10.6 (Snow Leopard) or 10.7 (Lion) as the oldest supported operating system release. In both cases, XCode must be installed along with the supplementary command line tools.

config-macos{10.6, 10.7}-gcc-{x86,x86_64,universal}.py: Create Intel 32 bit, 64 bit, or universal binaries using the `llvm-gcc` toolchain.

config-macos{10.6, 10.7}-icl-{x86,x86_64}.py: Create Intel 32 bit or 64 bit binaries using the Intel XE Composer toolchain. Versions XE 2012 and 2013 are known to work.

Note that the configuration files assume that XCode was installed in the `/Applications` folder. They must be be manually updated when this is not the case.

4.1.2. Selecting a configuration

Having chosen a configuration, copy it to the main directory and rename it to `config.py`, e.g.:

```
$ cp build/config-linux-gcc.py config.py
```

4.2. Compilation flags

There are several flags that affect the behavior of Mitsuba and must be specified at compile time. These usually don't need to be changed, but if you want to compile Mitsuba for spectral rendering, or to use double precision for internal computations then the following may be useful. Otherwise, you may skip ahead to the subsection that covers your operating system.

To change the compilation flags, open the `config.py` file that was just copied and look up the `CXXFLAG` parameter. The following options are available:

MTS_DEBUG Enable assertions etc. Usually a good idea, and enabled by default (even in release builds).

MTS_KD_DEBUG Enable additional checks in the kd-tree. This is quite slow and mainly useful to track down bugs when they are suspected.

MTS_KD_CONSERVE_MEMORY Use a more compact representation for triangle geometry (at the cost of speed). This flag causes Mitsuba to use the somewhat slower Moeller-Trumbore triangle intersection method instead of the default Wald intersection test, which has an overhead of 48 bytes per triangle. Off by default.

MTS_SSE Activate optimized SSE routines. On by default.

MTS_HAS_COHERENT_RT Include coherent ray tracing support (depends on `MTS_SSE`). This flag is activated by default.

MTS_DEBUG_FP Generated NaNs and overflows will cause floating point exceptions, which can be caught in a debugger. This is slow and mainly meant as a debugging tool for developers. Off by default.

SPECTRUM_SAMPLES=⟨..⟩ This setting defines the number of spectral samples (in the 368-830 *nm* range) that are used to render scenes. The default is 3 samples, in which case the renderer automatically turns into an RGB-based system. For high-quality spectral rendering, this should be set to 30 or higher. Refer also to Section 7.1.

SINGLE_PRECISION Do all computation in single precision. This is normally sufficient and therefore used as the default setting.

DOUBLE_PRECISION Do all computation in double precision. This flag is incompatible with `MTS_SSE`, `MTS_HAS_COHERENT_RT`, and `MTS_DEBUG_FP`.

MTS_GUI_SOFTWARE_FALLBACK Causes the GUI to use a software fallback instead of the hardware-accelerated realtime preview. This is useful when the binary will be executed over a remote link using a protocol such as RDP (which does not provide the requisite OpenGL features).

All of the default configurations files located in the `build` directory use the flags `SINGLE_PRECISION`, `SPECTRUM_SAMPLES=3`, `MTS_DEBUG`, `MTS_SSE`, as well as `MTS_HAS_COHERENT_RT`.

4.3. Building on Debian or Ubuntu Linux

You'll first need to install a number of dependencies. It is assumed here that you are using a recent version of Ubuntu Linux (Precise Pangolin / 12.04 LTS or later), hence some of the package may be named differently if you are using Debian Linux or another Ubuntu version.

First, run

```
$ sudo apt-get install build-essential scons mercurial qt4-dev-tools libpng12-dev
    libjpeg-dev libilmbase-dev libxerces-c-dev libboost-all-dev libopenexr-dev
    libglewmx-dev libxxf86vm-dev libpcrecpp0 libeigen3-dev libfftw3-dev
```

To get COLLADA support, you will also need to install the `collada-dom` packages or build them from scratch. Here, we install the `x86_64` binaries and development headers that can be found on the Mitsuba website (at `http://www.mitsuba-renderer.org/releases/current`)

```
$ sudo dpkg --install collada-dom_*.deb
```

To start a regular build, run

```
$ scons
```

inside the Mitsuba directory. In the case that you have multiple processors, you might want to parallelize the build by appending `-j` *core count* to the `scons` command. If all goes well, SCons should finish successfully within a few minutes:

```
scons: done building targets.
```

To run the renderer from the command line, you first have to import it into your shell environment:

```
$ source setpath.sh
```

Having set up everything, you can now move on to Section 5.

4.3.1. Creating Debian or Ubuntu Linux packages

The preferred way of redistristributing executables on Debian or Ubuntu Linux is to create `.deb` package files. To make custom Mitsuba packages, it is strongly recommended that you work with a pristine installation of the target operating system[3]. This can be done as follows: first, install `debootstrap` and download the most recent operating system release to a subdirectory. The following example is based on Ubuntu 12.04 LTS ("Precise Pangolin"), but the steps are almost identical for other versions of Ubuntu or Debian Linux.

```
$ sudo apt-get install debootstrap
$ sudo debootstrap --arch amd64 precise precise-pristine
```

Next, `chroot` into the created directory, enable the `multiverse` package repository, and install the necessary tools for creating package files:

```
$ sudo chroot precise-pristine
$ echo "deb http://archive.ubuntu.com/ubuntu precise universe" >> /etc/apt/sources.
    list
$ apt-get update
$ apt-get install debhelper dpkg-dev pkg-config
```

Now, you should be able to set up the remaining dependencies as described in Section 4.3. Once this is done, check out a copy of Mitsuba to the root directory of the `chroot` environment, e.g.

```
$ hg clone https://www.mitsuba-renderer.org/hg/mitsuba
```

To start the compilation process, enter

```
$ cd mitsuba
$ cp -R data/linux/debian debian
$ dpkg-buildpackage -nc
```

After everything has been built, you should find the created package files in the root directory.

4.3.2. Releasing Ubuntu packages

To redistribute Ubuntu packages over the Internet or a local network, it is convenient to put them into an `apt`-compatible repository. To prepare such a repository, put the two `deb`-files built in the last section, as well as the `collada-dom` `deb`-files into a public directory made available by a HTTP server and inside it, run

```
path-to-htdocs$ dpkg-scanpackages path/to/deb-directory /dev/null | gzip -9c >
    path/to/deb-directory/Packages.gz
```

This will create a respository index file named `Packages.gz`. Note that you must execute this command in the root directory of the HTTP server's web directory and provide the relative path to the package files – otherwise, the index file will specify the wrong package paths. Finally, the whole directory can be uploaded to some public location and then referenced by placing a line following the pattern

```
deb http://<path-to-deb-directory> ./
```

[3] Several commercial graphics drivers "pollute" the OpenGL setup so that the compiled Mitsuba binaries can only be used on machines using the same drivers. For this reason, it is better to work from a clean boostrapped install.

into the `/etc/apt/sources.list` file. This setup is convenient for distributing a custom Mitsuba build to many Debian or Ubuntu machines running (e.g. to nodes in a rendering cluster).

4.4. Building on Fedora Core

You'll first need to install a number of dependencies. It is assumed here that you are using FC15, hence some of the package may be named differently if you are using another version.

First, run

```
$ sudo yum install mercurial gcc-c++ scons boost-devel qt4-devel OpenEXR-devel
    xerces-c-devel python-devel glew-devel libpng-devel libjpeg-devel collada-dom-
    devel eigen3-devel fftw3-devel
```

Afterwards, simply run

```
$ scons
```

inside the Mitsuba directory. In the case that you have multiple processors, you might want to parallelize the build by appending `-j` *core count* to the command. If all goes well, SCons should finish successfully within a few minutes:

```
scons: done building targets.
```

To run the renderer from the command line, you first have to import it into your shell environment:

```
$ source setpath.sh
```

Having set up everything, you can now move on to Section 5.

4.4.1. Creating Fedora Core packages

To create `RPM` packages, you will need to install the `RPM` development tools:

```
$ sudo yum install rpmdevtools
```

Once this is done, run the following command in your home directory:

```
$ rpmdev-setuptree
```

and create a Mitsuba source package in the appropriate directory:

```
$ ln -s mitsuba mitsuba-0.5.0
$ tar czvf rpmbuild/SOURCES/mitsuba-0.5.0.tar.gz mitsuba-0.5.0/.
```

Finally, `rpmbuilder` can be invoked to create the package:

```
$ rpmbuild -bb mitsuba-0.5.0/data/linux/fedora/mitsuba.spec
```

After this command finishes, its output can be found in the directory `rpmbuild/RPMS`.

4.5. Building on Arch Linux

You'll first need to install a number of dependencies:

```
$ sudo pacman -S gcc xerces-c glew openexr boost libpng libjpeg qt scons mercurial
    python
```

For COLLADA support, you will also have to install the `collada-dom` library. For this, you can either install the binary package available on the Mitsuba website, or you can compile it yourself using the `PKGBUILD` supplied with Mitsuba, i.e.

```
$ cd <some-temporary-directory>
$ cp <path-to-mitsuba>/data/linux/arch/collada-dom/PKGBUILD .
$ makepkg PKGBUILD
<..compiling..>
$ sudo pacman -U <generated package file>
```

Finally, Eigen 3 must be installed. Again, there is a binary package on the Mitsuba website and the corresponding `PKGBUILD` can be obtained here: http://aur.archlinux.org/packages.php?ID=47884. Once all dependencies are taken care of, simply run

```
$ scons
```

inside the Mitsuba directory. In the case that you have multiple processors, you might want to parallelize the build by appending `-j` *core count* to the command. If all goes well, SCons should finish successfully within a few minutes:

```
scons: done building targets.
```

To run the renderer from the command line, you first have to import it into your shell environment:

```
$ source setpath.sh
```

Having set up everything, you can now move on to Section 5.

4.5.1. Creating Arch Linux packages

Mitsuba ships with a `PKGBUILD` file, which automatically builds a package from the most recent repository version:

```
$ makepkg data/linux/arch/mitsuba/PKGBUILD
```

4.6. Building on Windows

Compiling Mitsuba's dependencies on Windows is a laborious process; for convenience, there is a repository that provides them in precompiled form. To use this repository, clone it using Mercurial and rename the directory so that it forms the `dependencies` subdirectory inside the main Mitsuba directory, i.e. run something like

```
C:\>cd mitsuba
C:\mitsuba\>hg clone https://www.mitsuba-renderer.org/hg/dependencies_windows
C:\mitsuba\>rename dependencies_windows dependencies
```

There are a few other things that need to be set up: make sure that your installation of Visual Studio is up to date, since Mitsuba binaries created with versions prior to Service Pack 1 will crash.

Next, you will need to install Python 2.7.x (`www.python.org`) and SCons[4] (`http://www.scons.org`, any 2.x version will do) and ensure that they are contained in the `%PATH%` environment variable so that entering `scons` on the command prompt (`cmd.exe`) launches the build system.

[4] Note that on some Windows machines, the SCons installer generates a warning about not finding Python in the registry. In this case, you can instead run `python setup.py install` within the source release of SCons.

Having installed all dependencies, run the "Visual Studio *2010* Command Prompt" from the Start Menu (`x86` for 32-bit or `x64` for 64bit), navigate to the Mitsuba directory, and simply run

```
C:\mitsuba\>scons
```

In the case that you have multiple processors, you might want to parallelize the build by appending the option `-j` *core count* to the `scons` command.

If all goes well, the build process will finish successfully after a few minutes. *Note* that in comparison to the other platforms, you don't have to run the `setpath.sh` script at this point. All binaries are automatically copied into the `dist` directory, and they should be executed directly from there.

4.6.1. Integration with the Visual Studio interface

Basic Visual Studio 2010 integration with support for code completion exists for those who develop Mitsuba code on Windows. To use the supplied projects, simply double-click on one of the two files `build/mitsuba-msvc2010.sln` and `build/mitsuba-msvc2010.sln`. These Visual Studio projects still internally use the SCons-based build system to compile Mitsuba; whatever build configuration is selected within Visual Studio will be used to pick a matching configuration file from the `build` directory.

4.7. Building on Mac OS X

Remarks:

- Unfortunately, OpenMP is not available when compiling using the regular `clang` toolchain (it is available when using Intel XE Composer). This will cause the following parts of Mitsuba to run single-threaded: bitmap resampling (i.e. MIP map generation), blue noise point generation in the `dipole` plugin, as well as the `ppm` and `sppm` plugins.

Compiling Mitsuba's dependencies on Mac OS is a laborious process; for convenience, there is a repository that provides them in precompiled form. To use this repository, clone it using Mercurial and rename the directory so that it forms the `dependencies` subdirectory inside the main Mitsuba directory, i.e. run something like

```
$ cd mitsuba
$ hg clone https://www.mitsuba-renderer.org/hg/dependencies_macos
$ mv dependencies_macos dependencies
```

You will also need to install SCons (>2.0.0, available at `www.scons.org`) and a recent release of XCode, including its command-line compilation tools. Next, run

```
$ scons
```

inside the Mitsuba directory. In the case that you have multiple processors, you might want to parallelize the build by appending `-j` *core count* to the command. If all goes well, SCons should finish successfully within a few minutes:

```
scons: done building targets.
```

To run the renderer from the command line, you first have to import it into your shell environment:

```
$ source setpath.sh
```

5. Basic usage

The rendering functionality of Mitsuba can be accessed through a command line interface and an interactive Qt-based frontend. This section provides some basic instructions on how to use them.

5.1. Interactive frontend

To launch the interactive frontend, run `Mitsuba.app` on MacOS, `mtsgui.exe` on Windows, and `mtsgui` on Linux (after sourcing `setpath.sh`). You can also drag and drop scene files onto the application icon or the running program to open them. Two video tutorials on using the GUI can be found here: http://vimeo.com/13480342 (somewhat dated) and http://vimeo.com/50528092 (describing new features).

5.2. Command line interface

The `mitsuba` binary is an alternative non-interactive rendering frontend for command-line usage and batch job operation. To get a listing of the parameters it supports, run the executable without parameters:

```
$ mitsuba
```

Listing 1 shows the output resulting from this command. The most common mode of operation is to render a single scene, which is provided as a parameter, e.g.

```
$ mitsuba path-to/my-scene.xml
```

The next subsections explain various features of the `mitsuba` command line frontend.

5.2.1. Network rendering

Mitsuba can connect to network render nodes to parallelize a length rendering task over additional cores. To do this, pass a semicolon-separated list of machines to the `-c` parameter.

```
$ mitsuba -c machine1;machine2;... path-to/my-scene.xml
```

There are two different ways in which you can access render nodes:

- **Direct**: Here, you create a direct connection to a running `mtssrv` instance on another machine (`mtssrv` is the Mitsuba server process). From the the performance standpoint, this approach should always be preferred over the SSH method described below when there is a choice between them. There are some disadvantages though: first, you need to manually start `mtssrv` on every machine you want to use.

  And perhaps more importantly: the direct communication protocol makes no provisions for a malicious user on the remote side. It is too costly to constantly check the communication stream for illegal data sequences, so Mitsuba simply doesn't do it. The consequence of this is that you should only use the direct communication approach within trusted networks.

  For direct connections, you can specify the remote port as follows:

  ```
  $ mitsuba -c machine:1234 path-to/my-scene.xml
  ```

  When no port is explicitly specified, Mitsuba uses default value of 7554.

```
Mitsuba version 0.5.0, Copyright (c) 2014 Wenzel Jakob
Usage: mitsuba [options] <One or more scene XML files>
Options/Arguments:
   -h          Display this help text

   -D key=val  Define a constant, which can referenced as "$key" in the scene

   -o fname    Write the output image to the file denoted by "fname"

   -a p1;p2;.. Add one or more entries to the resource search path

   -p count    Override the detected number of processors. Useful for reducing
               the load or creating scheduling-only nodes in conjunction with
               the -c and -s parameters, e.g. -p 0 -c host1;host2;host3,...

   -q          Quiet mode - do not print any log messages to stdout

   -c hosts    Network rendering: connect to mtssrv instances over a network.
               Requires a semicolon-separated list of host names of the form
                       host.domain[:port] for a direct connection
                 or
                       user@host.domain[:path] for a SSH connection (where
                       "path" denotes the place where Mitsuba is checked
                       out -- by default, "~/mitsuba" is used)

   -s file     Connect to additional Mitsuba servers specified in a file
               with one name per line (same format as in -c)

   -j count    Simultaneously schedule several scenes. Can sometimes accelerate
               rendering when large amounts of processing power are available
               (e.g. when running Mitsuba on a cluster. Default: 1)

   -n name     Assign a node name to this instance (Default: host name)

   -t          Test case mode (see Mitsuba docs for more information)

   -x          Skip rendering of files where output already exists

   -r sec      Write (partial) output images every 'sec' seconds

   -b res      Specify the block resolution used to split images into parallel
               workloads (default: 32). Only applies to some integrators.

   -v          Be more verbose

   -w          Treat warnings as errors

   -z          Disable progress bars

 For documentation, please refer to http://www.mitsuba-renderer.org/docs.html
```

Listing 1: Command line options of the `mitsuba` binary

- **SSH**: This approach works as follows: The renderer creates a SSH connection to the remote side, where it launches a Mitsuba worker instance. All subsequent communication then passes through the encrypted link. This is completely secure but slower due to the encryption overhead. If you are rendering a complex scene, there is a good chance that it won't matter much since most time is spent doing computations rather than communicating

  Such an SSH link can be created simply by using a slightly different syntax:

  ```
  $ mitsuba -c username@machine path-to/my-scene.xml
  ```

  The above line assumes that the remote home directory contains a Mitsuba source directory named `mitsuba`, which contains the compiled Mitsuba binaries. If that is not the case, you need to provide the path to such a directory manually, e.g:

  ```
  $ mitsuba -c username@machine:/opt/mitsuba path-to/my-scene.xml
  ```

  For the SSH connection approach to work, you *must* enable passwordless authentication. Try opening a terminal window and running the command `ssh username@machine` (replace with the details of your remote connection). If you are asked for a password, something is not set up correctly — please see http://www.debian-administration.org/articles/152 for instructions.

  On Windows, the situation is a bit more difficult since there is no suitable SSH client by default. To get SSH connections to work, Mitsuba requires `plink.exe` (from PuTTY) to be on the path. For passwordless authentication with a Linux/OSX-based server, convert your private key to PuTTY's format using `puttygen.exe`. Afterwards, start `pageant.exe` to load and authenticate the key. All of these binaries are available from the PuTTY website.

  It is possible to mix the two approaches to access some machines directly and others over SSH.

When doing many network-based renders over the command line, it can become tedious to specify the connections every time. They can alternatively be loaded from a text file where each line contains a separate connection description as discussed previously:

```
$ mitsuba -s servers.txt path-to/my-scene.xml
```

where `servers.txt` e.g. contains

```
user1@machine1.domain.org:/opt/mitsuba
machine2.domain.org
machine3.domain.org:7346
```

5.2.2. Passing parameters

Any attribute in the XML-based scene description language (described in detail in Section 6) can be parameterized from the command line.

For instance, you can render a scene several times with different reflectance values on a certain material by changing its description to something like

```
<bsdf type="diffuse">
    <spectrum name="reflectance" value="$reflectance"/>
</bsdf>
```

and running Mitsuba as follows:

```
$ mitsuba -Dreflectance=0.1 -o ref_0.1.exr scene.xml
$ mitsuba -Dreflectance=0.2 -o ref_0.2.exr scene.xml
$ mitsuba -Dreflectance=0.5 -o ref_0.5.exr scene.xml
```

5.2.3. Writing partial images to disk

When doing lengthy command line renders on Linux or OSX, it is possible to send a signal to the process using

```
$ killall -HUP mitsuba
```

This causes the renderer to write out the partially finished image, after which it continues rendering. This can sometimes be useful to check if everything is working correctly.

5.2.4. Rendering an animation

The command line interface is ideally suited for rendering several files in batch operation. You can simply pass in the files using a wildcard in the filename.

If you've already rendered a subset of the frames and you only want to complete the remainder, add the `-x` flag, and all files with existing output will be skipped. You can also let the scheduler work on several scenes at once using the `-j` parameter—this is can accelerate parallelization over many machines: as some of the machines finish rendering the current frame, they can immediately start working on the next one instead of having to wait for all other cores to finish. Altogether, you might start the with parameters such as the following

```
$ mitsuba -xj 2 -c machine1;machine2;...  animation/frame_*.xml
```

Note that this requires a shell capable of expanding the asterisk into a list of filenames. The default Windows shell `cmd.exe` does not do this—however, the PowerShell supports the following syntax:

```
dir frame_*.xml | % { <path to mitsuba.exe> $_ }
```

5.3. Other programs

Mitsuba ships with a few other programs, which are explained in the remainder of this section.

5.3.1. Direct connection server

A Mitsuba compute node can be created using the `mtssrv` executable. By default, it will listen on port 7554:

```
$ mtssrv
..
maxwell: Listening on port 7554.. Send Ctrl-C or SIGTERM to stop.
```

Type `mtssrv -h` to see a list of available options. If you find yourself unable to connect to the server, `mtssrv` is probably listening on the wrong interface. In this case, please specify an explicit IP address or hostname:

```
$ mtssrv -i maxwell.cs.cornell.edu
```

As advised in Section 5.2, it is advised to run `mtssrv` *only* in trusted networks.

One nice feature of `mtssrv` is that it (like the `mitsuba` executable) also supports the `-c` and `-s` parameters, which create connections to additional compute servers. Using this feature, one can create hierarchies of compute nodes. For instance, the root `mttsrv` instance of such a hierarchy could share its work with a number of other machines running `mtssrv`, and each of these might also share their work with further machines, and so on...

The parallelization over such hierarchies happens transparently: when connecting a renderering process to the root node, it sees a machine with hundreds or thousands of cores, to which it can submit work without needing to worry about how exactly it is going to be spread out in the hierarchy.

Such hierarchies are mainly useful to reduce communication bottlenecks when distributing large resources (such as scenes) to remote machines. Imagine the following hypothetical scenario: you would like to render a 50MB-sized scene while at home, but rendering is too slow. You decide to tap into some extra machines available at your workplace, but this usually doesn't make things much faster because of the relatively slow broadband connection and the need to transmit your scene to every single compute node involved.

Using `mtssrv`, you can instead designate a central scheduling node at your workplace, which accepts connections and delegates rendering tasks to the other machines. In this case, you will only have to transmit the scene once, and the remaining distribution happens over the fast local network at your workplace.

5.3.2. Utility launcher

When working on a larger project, one often needs to implement various utility programs that perform simple tasks, such as applying a filter to an image or processing a matrix stored in a file. In a framework like Mitsuba, this unfortunately involves a significant coding overhead in initializing the necessary APIs on all supported platforms. To reduce this tedious work on the side of the programmer, Mitsuba comes with a utility launcher called `mtsutil`.

The general usage of this command is

```
$ mtsutil [options] <utility name> [arguments]
```

For a listing of all supported options and utilities, enter the command without parameters.

The second part of this manual explains how to develop such extensions yourself, specifically Section 12.

5.3.3. Tonemapper

One frequently used utility that shall be mentioned here is the batch tonemapper, which loads EXR/RGBE images and writes tonemapped 8-bit PNG/JPGs. This can save much time when one has to process many high dynamic-range images such as animation frames using the same basic operations, e.g. gamma correction, changing the overall brightness, resizing, cropping, etc. The available command line options are shown in Listing 2.

```
$ mtsutil tonemap
Synopsis: Loads one or more EXR/RGBE images and writes tonemapped 8-bit PNG/JPGs
Usage: mtsutil tonemap [options] <EXR/RGBE file (s)>
Options/Arguments:
   -h             Display this help text

   -g gamma       Specify the gamma value (The default is -1 => sRGB)

   -m multiplier  Multiply the pixel values by 'multiplier' (Default = 1)

   -b r,g,b       Color balance: apply the specified per-channel multipliers

   -c x,y,w,h     Crop: tonemap a given rectangle instead of the entire image

   -s w,h         Resize the output image to the specified resolution

   -r x,y,w,h,i   Add a rectangle at the specified position and intensity, e.g.
                  to make paper figures. The intensity should be in [0, 255].

   -f fmt         Request a certain output format (png/jpg, default:png)

   -a             Require the output image to have an alpha channel

   -p key,burn    Run Reinhard et al.'s photographic tonemapping operator. 'key'
                  between [0, 1] chooses between low and high-key images and
                  'burn' (also [0, 1]) controls how much highlights may burn out

   -B fov         Apply a bloom filter that simulates scattering in the human
                  eye. Requires the approx. field of view of the images to be
                  processed in order to compute a point spread function.

   -x             Temporal coherence mode: activate this flag when tonemapping
                  frames of an animation using the '-p' option to avoid flicker

   -o file        Save the output with a given filename

   -t             Multithreaded: process several files in parallel

 The operations are ordered as follows: 1. crop, 2. bloom, 3. resize, 4. color
 balance, 5. tonemap, 6. annotate. To simply process a directory full of EXRs
 in parallel, run the following: 'mtsutil tonemap -t path-to-directory/*.exr'
```

Listing 2: Command line options of the `mtsutil tonemap` utility

6. Scene file format

Mitsuba uses a very simple and general XML-based format to represent scenes. Since the framework's philosophy is to represent discrete blocks of functionality as plugins, a scene file can essentially be interpreted as description that determines which plugins should be instantiated and how they should interface with each other. In the following, we'll look at a few examples to get a feeling for the scope of the format.

A simple scene with a single mesh and the default lighting and camera setup might look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<scene version="0.5.0">
    <shape type="obj">
        <string name="filename" value="dragon.obj"/>
    </shape>
</scene>
```

The scene version attribute denotes the release of Mitsuba that was used to create the scene. This information allows Mitsuba to always correctly process the file irregardless of any potential future changes in the scene description language.

This example already contains the most important things to know about format: you can have *objects* (such as the objects instantiated by the `scene` or `shape` tags), which are allowed to be nested within each other. Each object optionally accepts *properties* (such as the `string` tag), which further characterize its behavior. All objects except for the root object (the `scene`) cause the renderer to search and load a plugin from disk, hence you must provide the plugin name using `type=".."` parameter.

The object tags also let the renderer know *what kind* of object is to be instantiated: for instance, any plugin loaded using the `shape` tag must conform to the *Shape* interface, which is certainly the case for the plugin named `obj` (it contains a WaveFront OBJ loader). Similarly, you could write

```
<?xml version="1.0" encoding="utf-8"?>
<scene version="0.5.0">
    <shape type="sphere">
        <float name="radius" value="10"/>
    </shape>
</scene>
```

This loads a different plugin (`sphere`) which is still a *Shape*, but instead represents a sphere configured with a radius of 10 world-space units. Mitsuba ships with a large number of plugins; please refer to the next chapter for a detailed overview of them.

The most common scene setup is to declare an integrator, some geometry, a sensor (e.g. a camera), a film, a sampler and one or more emitters. Here is a more complex example:

```
<?xml version="1.0" encoding="utf-8"?>

<scene version="0.5.0">
    <integrator type="path">
        <!-- Path trace with a max. path length of 8 -->
        <integer name="maxDepth" value="8"/>
    </integrator>
```

```
    <!-- Instantiate a perspective camera with 45 degrees field of view -->
    <sensor type="perspective">
        <!-- Rotate the camera around the Y axis by 180 degrees -->
        <transform name="toWorld">
            <rotate y="1" angle="180"/>
        </transform>
        <float name="fov" value="45"/>

        <!-- Render with 32 samples per pixel using a basic
             independent sampling strategy -->
        <sampler type="independent">
            <integer name="sampleCount" value="32"/>
        </sampler>

        <!-- Generate an EXR image at HD resolution -->
        <film type="hdrfilm">
            <integer name="width" value="1920"/>
            <integer name="height" value="1080"/>
        </film>
    </sensor>

    <!-- Add a dragon mesh made of rough glass (stored as OBJ file) -->
    <shape type="obj">
        <string name="filename" value="dragon.obj"/>

        <bsdf type="roughdielectric">
            <!-- Tweak the roughness parameter of the material -->
            <float name="alpha" value="0.01"/>
        </bsdf>
    </shape>

    <!-- Add another mesh -- this time, stored using Mitsuba's own
         (compact) binary representation -->
    <shape type="serialized">
        <string name="filename" value="lightsource.serialized"/>
        <transform name="toWorld">
            <translate x="5" y="-3" z="1"/>
        </transform>

        <!-- This mesh is an area emitter -->
        <emitter type="area">
            <rgb name="radiance" value="100,400,100"/>
        </emitter>
    </shape>
</scene>
```

This example introduces several new object types (`integrator`, `sensor`, `bsdf`, `sampler`, `film`, and `emitter`) and property types (`integer`, `transform`, and `rgb`). As you can see in the example, objects are usually declared at the top level except if there is some inherent relation that links them to another object. For instance, BSDFs are usually specific to a certain geometric object, so they appear as a child object of a shape. Similarly, the sampler and film affect the way in which rays are generated

from the sensor and how it records the resulting radiance samples, hence they are nested inside it.

6.1. Property types

This section documents all of the ways in which properties can be supplied to objects. If you are more interested in knowing which properties a certain plugin accepts, you should look at the next section instead.

6.1.1. Numbers

Integer and floating point values can be passed as follows:

```
<integer name="intProperty" value="1234"/>
<float name="floatProperty" value="1.234"/>
<float name="floatProperty2" value="-1.5e3"/>
```

Note that you must adhere to the format expected by the object, i.e. you can't pass an integer property to an object, which expects a floating-point value associated with that name.

6.1.2. Strings

Passing strings is straightforward:

```
<string name="stringProperty" value="This is a string"/>
```

6.1.3. RGB color values

In Mitsuba, color data is specified using the <rgb>, and <srgb>, or <spectrum> tags. We begin with the first two, which are most commonly used. The RGB tags expect red, green, and blue color values in floating point format, which are usually between 0 and 1 when specifying reflectance values. The <srgb> tag internally causes the specified value to be linearized by mapping it through an inverse sRGB gamma curve:

```
<rgb name="spectrumProperty" value="0.2, 0.8, 0.4"/>
<srgb name="spectrumProperty" value="0.4, 0.3, 0.2"/>
```

The <srgb> tag also accepts HTML-style hex values, e.g.:

```
<srgb name="spectrumProperty" value="#f9aa34"/>
```

When Mitsuba is compiled with the default settings, it internally uses linear RGB to represent colors, so these values are directly used. However, when configured for spectral rendering [5], a color spectrum that has a matching RGB value must be found. This is a classic underdetermined problem, since there is an infinite number of spectra corresponding to any particular RGB value.

Mitsuba relies on a method by Smits et al. [42] to find a smooth and physically "plausible" spectrum. To do so, it chooses one of two variants of Smits' approach depending on whether the spectrum contains a unitless reflectance value, or a radiance-valued intensity. This choice can be enforced via the `intent` XML attribute, e.g.:

[5] Note that the official releases all use linear RGB—to do spectral renderings, you will have to compile Mitsuba yourself.

```
<rgb name="spectrumProperty" intent="reflectance" value="0.2, 0.8, 0.4"/>
<rgb name="spectrumProperty" intent="illuminant" value="0.2, 0.8, 0.4"/>
```

Usually this attribute is not neccessary: Mitsuba detects when an RGB value is specified in the declaration of a light source and uses `intent="illuminant"` in this case and `intent="reflectance"` everywhere else.

6.1.4. Color spectra

Mitsuba can also work with spectral color data. The exact internal representation of such spectra depends on how the renderer was compiled (see Section 4.2 for details).

When `SPECTRUM_SAMPLES` was set 3 at compile time (the default for the official builds), Mistuba uses a basic linear RGB representation and thus always converts color spectra to RGB. For other values (e.g. `SPECTRUM_SAMPLES`=20), then renderer performs all internal computations using a full spectral representation with the specified number of bins.

The preferred way of passing color spectra to the renderer is to explicitly denote the associated wavelengths of each value:

```
<spectrum name="spectrumProperty" value="400:0.56, 500:0.18, 600:0.58, 700:0.24"/>
```

This is a mapping from wavelength in nanometers (before the colon) to a reflectance or intensity value (after the colon). Values in between are linearly interpolated from the two closest neighbors.

A useful shortcut to get a "white" or uniform spectrum, it is to provide only a single value:

```
<spectrum name="spectrumProperty" value="0.56"/>
```

The exact definition a white spectrum depends on whether it specifies a unitless reflectance value or a radiance-valued intensity. As before, Mitsuba tries to detect this automatically depending on whether or not the <spectrum> tag occurs within a light source declaration, and the `intent` attribute can be used to override the default behavior. In particular, the next snippet creates a uniform spectrum:

```
<spectrum name="spectrumProperty" intent="reflectance" value="0.56"/>
```

On the other hand, the following creates a multiple of the white point (the CIE D65 illuminant):

```
<spectrum name="spectrumProperty" intent="illuminant" value="0.56"/>
```

Another (discouraged) option is to directly provide the spectrum in Mitsuba's internal representation, avoiding the need for any kind of conversion. However, this is problematic, since the associated scene will not work when Mitsuba is compiled with a different value of `SPECTRUM_SAMPLES`. For completeness, the possibility is explained nonetheless. Assuming that the 360-830nm range is discretized into ten 47nm-sized blocks (i.e. `SPECTRUM_SAMPLES` is set to 10), their values can be specified as

```
<spectrum name="spectrumProperty" value=".2, .2, .8, .4, .6, .5, .1, .9, .4, .2"/>
```

When spectral power or reflectance distributions are obtained from measurements (e.g. at 10nm intervals), they are usually quite unwiedy and can clutter the scene description. For this reason, there is yet another way to pass a spectrum by loading it from an external file:

```
<spectrum name="spectrumProperty" filename="measuredSpectrum.spd"/>
```

The file should contain a single measurement per line, with the corresponding wavelength in nanometers and the measured value separated by a space. Comments are allowed. Here is an example:

```
# This file contains a measured spectral power/reflectance distribution
406.13 0.703313
413.88 0.744563
422.03 0.791625
430.62 0.822125
435.09 0.834000
...
```

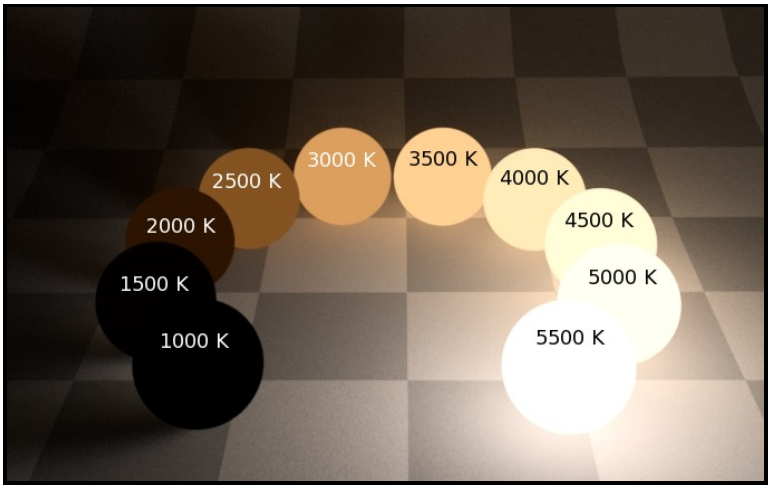


Figure 1: A few simulated black body emitters over a range of temperature values

Finally, it is also possible to specify the spectral distribution of a black body emitter (Figure 1), where the temperature is given in Kelvin.

```
<blackbody name="spectrumProperty" temperature="5000K"/>
```

Note that attaching a black body spectrum to the `intensity` property of a emitter introduces physical units into the rendering process of Mitsuba, which is ordinarily a unitless system[6].

Specifically, the black body spectrum has units of power (W) per unit area (m^{-2}) per steradian (sr^{-1}) per unit wavelength (nm^{-1}). If these units are inconsistent with your scene description (e.g. because it is modeled in millimeters or kilometers), you may use the optional `scale` attribute to adjust them, e.g.:

```
<!-- Scale black body radiance by a factor of 1/1000 -->
<blackbody name="spectrumProperty" temperature="5000K" scale="1e-3"/>
```

6.1.5. Vectors, Positions

Points and vectors can be specified as follows:

```
<point name="pointProperty" x="3" y="4" z="5"/>
<vector name="vectorProperty" x="3" y="4" z="5"/>
```

It is important that whatever you choose as world-space units (meters, inches, etc.) is used consistently in all places.

[6] This means that the units of pixel values in a rendering are completely dependent on the units of the user input, including the unit of world-space distance and the units of the light source emission profile.

6.1.6. Transformations

Transformations are the only kind of property that require more than a single tag. The idea is that, starting with the identity, one can build up a transformation using a sequence of commands. For instance, a transformation that does a translation followed by a rotation might be written like this:

```
<transform name="trafoProperty">
    <translate x="-1" y="3" z="4"/>
    <rotate y="1" angle="45"/>
</transform>
```

Mathematically, each incremental transformation in the sequence is left-multiplied onto the current one. The following choices are available:

- Translations, e.g.

  ```
  <translate x="-1" y="3" z="4"/>
  ```

- Counter-clockwise rotations around a specified axis. The angle is given in degrees, e.g.

  ```
  <rotate x="0.701" y="0.701" z="0" angle="180"/>
  ```

- Scaling operations. The coefficients may also be negative to obtain a flip, e.g.

  ```
  <scale value="5"/>           <!-- uniform scale -->
  <scale x="2" y="1" z="-1"/>  <!-- non-unform scale -->
  ```

- Explicit 4×4 matrices, e.g

  ```
  <matrix value="0 -0.53 0 -1.79 0.92 0 0 8.03 0 0 0.53 0 0 0 0 1"/>
  ```

- `lookat` transformations — this is primarily useful for setting up cameras (and spot lights). The `origin` coordinates specify the camera origin, `target` is the point that the camera will look at, and the (optional) `up` parameter determines the "upward" direction in the final rendered image. The `up` parameter is not needed for spot lights.

  ```
  <lookat origin="10, 50, -800" target="0, 0, 0" up="0, 1, 0"/>
  ```

Cordinates that are zero (for `translate` and `rotate`) or one (for `scale`) do not explicitly have to be specified.

6.2. Animated transformations

Most shapes, emitters, and sensors in Mitsuba can accept both normal transformations and *animated transformations* as parameters. The latter is useful to render scenes involving motion blur (Figure 2). The syntax used to specify these is slightly different:

```
<animation name="trafoProperty">
  <transform time="0">
     .. chained list of transformations as discussed above ..
  </transform>

  <transform time="1">
     .. chained list of transformations as discussed above ..
  </transform>

  .. additional transformations (optional) ..
</animation>
```

Figure 2: Beware the dragon: a triangle mesh undergoing linear motion with several keyframes (object courtesy of XYZRGB)

Mitsuba then decomposes each transformation into a scale, translation, and rotation component and interpolates[7] these for intermediate time values. It is important to specify appropriate shutter open/close times to the sensor so that the motion is visible.

6.3. References

Quite often, you will find yourself using an object (such as a material) in many places. To avoid having to declare it over and over again, which wastes memory, you can make use of references. Here is an example of how this works:

[7] Using linear interpolation for the scale and translation component and spherical linear quaternion interpolation for the rotation component.

```
<scene version="0.5.0">
    <texture type="bitmap" id="myImage">
        <string name="filename" value="textures/myImage.jpg"/>
    </texture>

    <bsdf type="diffuse" id="myMaterial">
        <!-- Reference the texture named myImage and pass it
            to the BRDF as the reflectance parameter -->
        <ref name="reflectance" id="myImage"/>
    </bsdf>

    <shape type="obj">
        <string name="filename" value="meshes/myShape.obj"/>

        <!-- Reference the material named myMaterial -->
        <ref id="myMaterial"/>
    </shape>
</scene>
```

By providing a unique `id` attribute in the object declaration, the object is bound to that identifier upon instantiation. Referencing this identifier at a later point (using the `<ref id="..."/>` tag) will add the instance to the parent object, with no further memory allocation taking place. Note that some plugins expect their child objects to be named[8]. For this reason, a name can also be associated with the reference.

Note that while this feature is meant to efficiently handle materials, textures, and participating media that are referenced from multiple places, it cannot be used to instantiate geometry—if this functionality is needed, take a look at the `instance` plugin.

6.4. Including external files

A scene can be split into multiple pieces for better readability. to include an external file, please use the following command:

```
<include filename="nested-scene.xml"/>
```

In this case, the file `nested-scene.xml` must be a proper scene file with a `<scene>` tag at the root. This feature is sometimes very convenient in conjunction with the `-D key=value` flag of the `mitsuba` command line renderer (see the previous section for details). This lets you include different parts of a scene configuration by changing the command line parameters (and without having to touch the XML file):

```
<include filename="nested-scene-$version.xml"/>
```

6.5. Default parameter values

As mentioned before, scenes may contain named parameters that are supplied via the command line:

[8] For instance, material plugins such as `diffuse` require that nested texture instances explicitly specify the parameter to which they want to bind (e.g. "`reflectance`").

```
<bsdf type="diffuse">
    <rgb name="reflectance" value="$reflectance"/>
</bsdf>
```

In this case, an error will occur when loading the scene without an explicit command line argument of the form `-Dreflectance=`⟨*something*⟩. For convenience, it is possible to specify a default parameter value that takes precedence when no command line arguments are given. The syntax for this is

```
<default name="reflectance" value="something"/>
```

and must precede occurrences of the parameter in the XML file.

6.6. Aliases

Sometimes, it can be useful to associate an object (e.g. a scattering model) with multiple identifiers. This can be accomplished using the `alias as=..` keyword:

```
<bsdf type="diffuse" id="myMaterial1"/>
<alias id="myMaterial1" as="myMaterial2"/>
```

After this statement, the diffuse scattering model will be bound to *both* identifiers "`myMaterial1`" and "`myMaterial2`".

7. Miscellaneous topics

7.1. A word about color spaces

When using one of the downloadable release builds of Mitsuba, or a version that was compiled with the default settings, the renderer internally operates in *RGB mode*: all computations are performed using a representation that is based on the three colors red, green, and blue.

More specifically, these are the intensities of the red, green, and blue primaries defined by the sRGB standard (ITU-R Rec. BT. 709-3 primaries with a D65 white point). Mitsuba transparently converts all input data (e.g. textures) into this space before rendering. This is an intuitive default which yields fast computations and satisfactory results for most applications.

Low dynamic range images exported using the `ldrfilm` will be stored in a sRGB-compatible format that accounts for the custom gamma curves mandated by this standard. They should display as intended across a wide range of display devices.

When saving high dynamic range output (e.g. OpenEXR, RGBE, or PFM), the computed radiance values are exported in a linear form (i.e. without having the sRGB gamma curve applied to it), which is the most common way of storing high dynamic range data. It is important to keep in mind that other applications may not support this "linearized sRGB" space—in particular, the Mac OS preview currently does not display images with this encoding correctly.

7.1.1. Spectral rendering

Some predictive rendering applications will require a more realistic space for interreflection computations. In such cases, Mitsuba can be switched to *spectral mode*. This can be done by compiling it with the `SPECTRUM_SAMPLES`=*n* parameter (Section 4), where *n* is usually between 15 and 30.

Now, all input parameters are converted into color spectra with the specified number of discretizations, and the computation then proceeds using this space. The process of writing an output image works differently: when spectral output is desired (`hdrfilm`, `tiledhdrfilm`, and `mfilm` support this), Mitsuba creates special image files with many color channels (one per spectral band). Generally, other applications will not be able to display these images. The Mitsuba GUI can be used to view them, however (simply drag & drop an image onto the application).

It is also possible to write out XYZ tristimulus values, in which case the spectral data is convolved with the CIE 1931 color matching curves. This is most useful to users who want to do their own color processing in a space with a wide gamut.

Finally, sRGB output is still possible. However, the color processing used in this case is fairly naïve: out-of-gamut values are simply clipped. This is something that may be improved in the future (e.g. by making use of a color management library like `lcms2`)

7.2. Using Mitsuba from Makefiles

Sometimes it is useful to run `mitsuba` from a standard Unix Makefile. This is a bit inconvenient because shell commands in Makefiles are executed using the classic `sh` shell, which is incompatible with the `setpath.sh` script. A simple workaround in this case is to explicitly call `bash` or `zsh`, e.g.

```
MITSUBA_HOME=⟨..⟩
%.exr: %.xml
    bash -c ". $(MITSUBA_HOME)/setpath.sh; mitsuba -o $@ $<"
```

8. Plugin reference

The following subsections describe the available Mitsuba plugins, usually along with example renderings and a description of what each parameter does. They are separated into subsections covering textures, surface scattering models, etc.

Each subsection begins with a brief general description. The documentation of a plugin always starts on a new page and is preceded by a table similar to the one below:

Parameter	Type	Description
`softRays`	`boolean`	Try not to damage objects in the scene by shooting softer rays (Default: `false`)
`darkMatter`	`float`	Controls the proportionate amount of dark matter present in the scene. (Default: 0.83)

Suppose this hypothetical plugin is an *integrator* named `amazing`. Then, based on this description, it can be instantiated from an XML scene file using a custom configuration such as:

```
<integrator type="amazing">
    <boolean name="softerRays" value="true"/>
    <float name="darkMatter" value="0.4"/>
</integrator>
```

In some cases[9], plugins also indicate that they accept nested plugins as input arguments. These can either be *named* or *unnamed*. If the `amazing` integrator also accepted the following two parameters

Parameter	Type	Description
(Nested plugin)	`integrator`	A nested integrator which does the actual hard work
`puppies`	`texture`	This must be used to supply a cute picture of puppies

then it can be instantiated e.g. as follows

```
<integrator type="amazing">
    <boolean name="softerRays" value="true"/>
    <float name="darkMatter" value="0.4"/>
    <integrator type="path"/>
    <texture name="puppies" type="bitmap">
        <string name="filename" value="cute.jpg"/>
    </texture>
</integrator>
```

or, if these were already instantiated previously and are now bound to the *identifiers* (Section 6) `myPathTracer` and `myTexture`, the following also works:

```
<integrator type="amazing">
    <boolean name="softerRays" value="true"/>
    <float name="darkMatter" value="0.4"/>
    <ref id="myPathTracer"/>
    <ref name="puppies" id="myTexture"/>
</integrator>
```

[9] Note that obvious parameters are generally omitted. For instance, all shape plugins accept a surface scattering plugin, but this is left out from the documentation for brevity.

8.1. Shapes

This section presents an overview of the shape plugins that are released along with the renderer.

In Mitsuba, shapes define surfaces that mark transitions between different types of materials. For instance, a shape could describe a boundary between air and a solid object, such as a piece of rock. Alternatively, a shape can mark the beginning of a region of space that isn't solid at all, but rather contains a participating medium, such as smoke or steam. Finally, a shape can be used to create an object that emits light on its own.

Shapes are usually declared along with a surface scattering model (named "BSDF", see Section 8.2 for details). This BSDF characterizes what happens *at the surface*. In the XML scene description language, this might look like the following:

```
<scene version="0.5.0">
    <shape type="... shape type ...">
        ... shape parameters ...

        <bsdf type="... bsdf type ...">
            ... bsdf parameters ..
        </bsdf>

        <!-- Alternatively: reference a named BSDF that
             has been declared previously

             <ref id="myBSDF"/>
        -->
    </shape>
</scene>
```

When a shape marks the transition to a participating medium (e.g. smoke, fog, ..), it is furthermore necessary to provide information about the two media that lie at the *interior* and *exterior* of the shape. This informs the renderer about what happens in the region of space *surrounding the surface*.

```
<scene version="0.5.0">
    <shape type="... shape type ...">
        ... shape parameters ...

        <medium name="interior" type="... medium type ...">
            ... medium parameters ...
        </medium>

        <medium name="exterior" type="... medium type ...">
            ... medium parameters ...
        </medium>

        <!-- Alternatively: reference named media that
             have been declared previously

             <ref name="interior" id="myMedium1"/>
             <ref name="exterior" id="myMedium2"/>
        -->
    </shape>
</scene>
```

You may have noticed that the previous XML example dit not make any mention of surface scattering models (BSDFs). In Mitsuba, such a shape declaration creates an *index-matched* boundary. This means that incident illumination will pass through the surface without undergoing any kind of interaction. However, the renderer will still uses the information available in the shape to correctly account for the medium change.

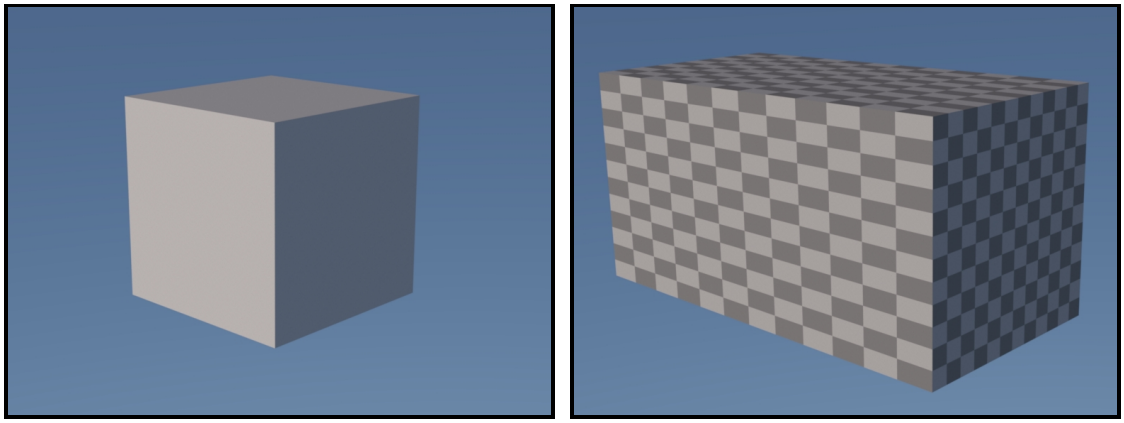
It is also possible to create *index-mismatched* boundaries between media, where some of the light is affected by the boundary transition:

```
<scene version="0.5.0">
    <shape type="... shape type ...">
        ... shape parameters ...

        <bsdf type="... bsdf type ...">
            ... bsdf parameters ..
        </bsdf>

        <medium name="interior" type="... medium type ...">
            ... medium parameters ...
        </medium>

        <medium name="exterior" type="... medium type ...">
            ... medium parameters ...
        </medium>

        <!-- Alternatively: reference named media and BSDF
             instances that have been declared previously

             <ref id="myBSDF"/>
             <ref name="interior" id="myMedium1"/>
             <ref name="exterior" id="myMedium2"/>
        -->
    </shape>
</scene>
```

This constitutes the standard ways in which a shape can be declared. The following subsections discuss the available types in greater detail.

8.1.1. Cube intersection primitive (cube)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional linear object-to-world transformation. (Default: none (i.e. object space = world space))
`flipNormals`	`boolean`	Is the cube inverted, i.e. should the normal vectors be flipped? (Default: `false`, i.e. the normals point outside)



(a) Basic example (b) A textured and stretched cube with the default parameterization (Listing 3)

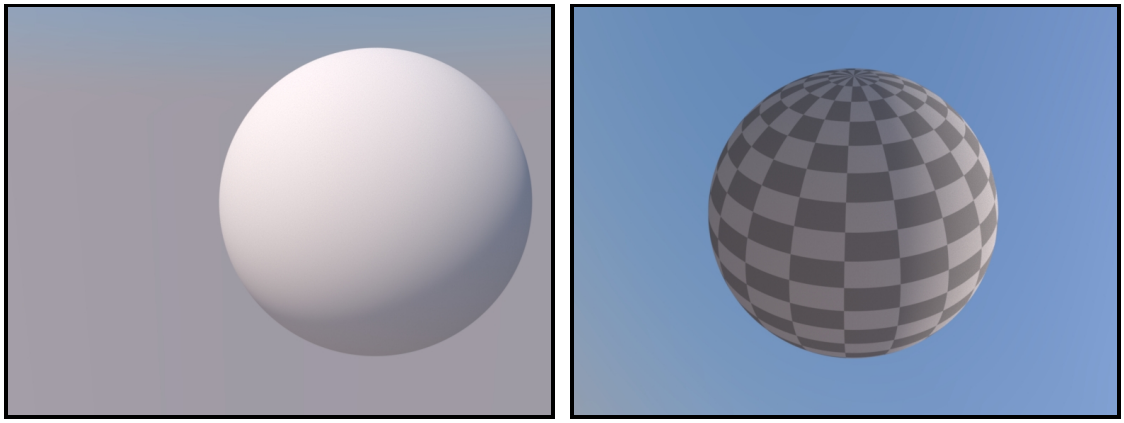
This shape plugin describes a simple cube/cuboid intersection primitive. By default, it creates a cube between the world-space positions $(-1,-1,-1)$ and $(1,1,1)$. However, an arbitrary linear transformation may be specified to translate, rotate, scale or skew it as desired. The parameterization of this shape maps every face onto the rectangle $[0,1]^2$ in uv space.

```
<shape type="cube">
  <transform name="toWorld">
    <scale z="2"/>
  </transform>

  <bsdf type="diffuse">
    <texture type="checkerboard" name="reflectance">
      <float name="uvscale" value="6"/>
    </texture>
  </bsdf>
</shape>
```

Listing 3: Example of a textured and stretched cube

8.1.2. Sphere intersection primitive (`sphere`)

Parameter	Type	Description
`center`	`point`	Center of the sphere in object-space (Default: (0, 0, 0))
`radius`	`float`	Radius of the sphere in object-space units (Default: 1)
`toWorld`	`transform` or `animation`	Specifies an optional linear object-to-world transformation. Note that non-uniform scales are not permitted! (Default: none (i.e. object space = world space))
`flipNormals`	`boolean`	Is the sphere inverted, i.e. should the normal vectors be flipped? (Default: `false`, i.e. the normals point outside)



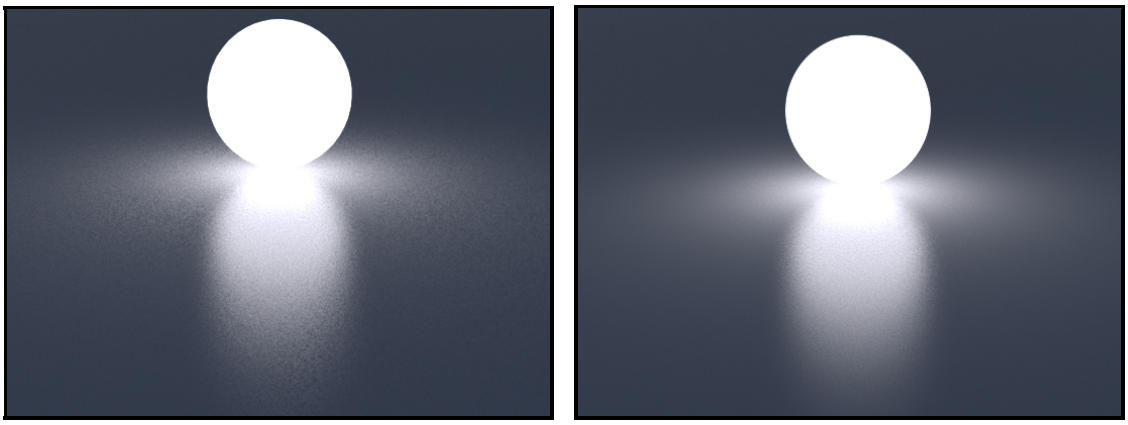
(a) Basic example, see Listing 4 (b) A textured sphere with the default parameterization

This shape plugin describes a simple sphere intersection primitive. It should always be preferred over sphere approximations modeled using triangles.

```
<shape type="sphere">
    <transform name="toWorld">
        <scale value="2"/>
        <translate x="1" y="0" z="0"/>
    </transform>
    <bsdf type="diffuse"/>
</shape>

<shape type="sphere">
    <point name="center" x="1" y="0" z="0"/>
    <float name="radius" value="2"/>
    <bsdf type="diffuse"/>
</shape>
```

Listing 4: A sphere can either be configured using a linear `toWorld` transformation or the `center` and `radius` parameters (or both). The above two declarations are equivalent.

When a `sphere` shape is turned into an `area` light source, Mitsuba switches to an efficient sampling strategy [41] that has particularly low variance. This makes it a good default choice for lighting new scenes (Figure 3).



(a) Spherical area light modeled using triangles (b) Spherical area light modeled using the `sphere` plugin

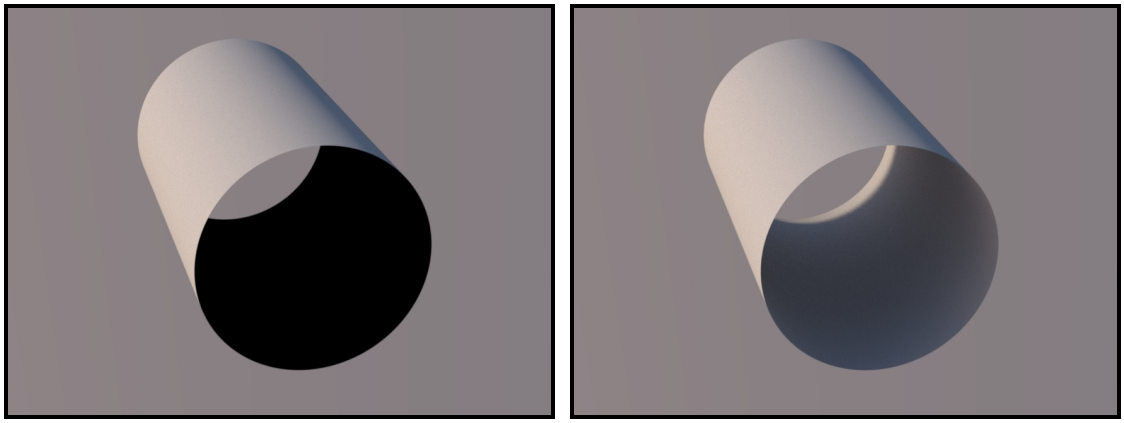
Figure 3: Area lights built from the combination of the `area` and `sphere` plugins produce renderings that have an overall lower variance.

```
<shape type="sphere">
    <point name="center" x="0" y="1" z="0"/>
    <float name="radius" value="1"/>

    <emitter type="area">
        <blackbody name="intensity" temperature="7000K"/>
    </emitter>
</shape>
```

Listing 5: Instantiation of a sphere emitter

8.1.3. Cylinder intersection primitive (`cylinder`)

Parameter	Type	Description
`p0`	`point`	Object-space starting point of the cylinder's centerline (Default: (0, 0, 0))
`p1`	`point`	Object-space endpoint of the cylinder's centerline (Default: (0, 0, 1))
`radius`	`float`	Radius of the cylinder in object-space units (Default: 1)
`flipNormals`	`boolean`	Is the cylinder inverted, i.e. should the normal vectors be flipped? (Default: `false`, i.e. the normals point outside)
`toWorld`	`transform` or `animation`	Specifies an optional linear object-to-world transformation. Note that non-uniform scales are not permitted! (Default: none (i.e. object space = world space))



(a) Cylinder with the default one-sided shading (b) Cylinder with two-sided shading, see Listing 6

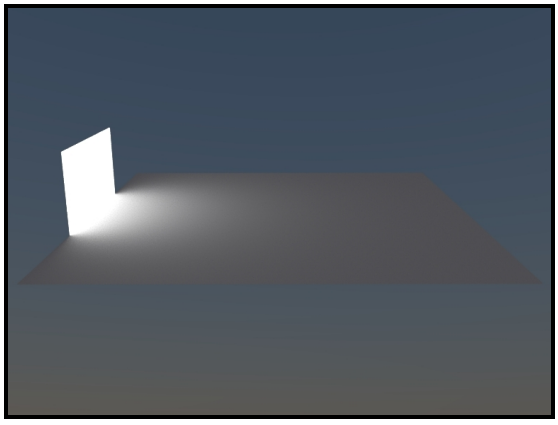
This shape plugin describes a simple cylinder intersection primitive. It should always be preferred over approximations modeled using triangles. Note that the cylinder does not have endcaps – also, it's interior has inward-facing normals, which most scattering models in Mitsuba will treat as fully absorbing. If this is not desirable, consider using the `twosided` plugin.

```
<shape type="cylinder">
    <float name="radius" value="0.3"/>
    <bsdf type="twosided">
        <bsdf type="diffuse"/>
    </bsdf>
</shape>
```

Listing 6: A simple example for instantiating a cylinder, whose interior is visible

8.1.4. Rectangle intersection primitive (`rectangle`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies a linear object-to-world transformation. It is allowed to use non-uniform scaling, but no shear. (Default: none (i.e. object space = world space))
`flipNormals`	`boolean`	Is the rectangle inverted, i.e. should the normal vectors be flipped? (Default: `false`)



(a) Two rectangles configured as a reflective surface and an emitter (Listing 7)

This shape plugin describes a simple rectangular intersection primitive. It is mainly provided as a convenience for those cases when creating and loading an external mesh with two triangles is simply too tedious, e.g. when an area light source or a simple ground plane are needed.

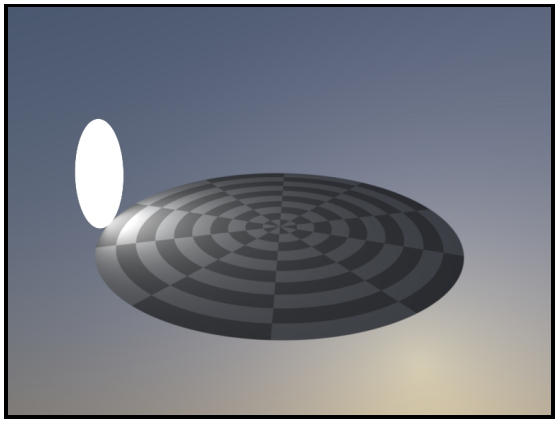
By default, the rectangle covers the XY-range $[-1, 1] \times [-1, 1]$ and has a surface normal that points into the positive Z direction. To change the rectangle scale, rotation, or translation, use the `toWorld` parameter.

```
<scene version="0.5.0">
    <shape type="rectangle">
        <bsdf type="diffuse"/>
    </shape>
    <shape type="rectangle">
        <transform name="toWorld">
            <rotate x="1" angle="90"/>
            <scale x="0.4" y="0.3" z="0.2"/>
            <translate y="1" z="0.2"/>
        </transform>
        <emitter type="area">
            <spectrum name="intensity" value="3"/>
        </emitter>
    </shape>
    <!-- ... other definitions ... -->
</scene>
```

Listing 7: A simple example involving two rectangle instances

8.1.5. Disk intersection primitive (`disk`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies a linear object-to-world transformation. Note that non-uniform scales are not permitted! (Default: none (i.e. object space = world space))
`flipNormals`	`boolean`	Is the disk inverted, i.e. should the normal vectors be flipped? (Default: `false`)



(a) Rendering with an disk emitter and a textured disk, showing the default parameterization. (Listing 8)

This shape plugin describes a simple disk intersection primitive. It is usually preferable over discrete approximations made from triangles.

By default, the disk has unit radius and is located at the origin. Its surface normal points into the positive Z direction. To change the disk scale, rotation, or translation, use the `toWorld` parameter.

```
<scene version="0.5.0">
    <shape type="disk">
        <bsdf type="diffuse">
            <texture name="reflectance" type="checkerboard">
                <float name="uvscale" value="5"/>
            </texture>
        </bsdf>
    </shape>
    <shape type="disk">
        <transform name="toWorld">
            <rotate x="1" angle="90"/>
            <scale value="0.3"/>
            <translate y="1" z="0.3"/>
        </transform>
        <emitter type="area">
            <spectrum name="intensity" value="4"/>
        </emitter>
    </shape>
</scene>
```

Listing 8: A simple example involving two disk instances

8.1.6. Wavefront OBJ mesh loader (`obj`)

Parameter	Type	Description
`filename`	`string`	Filename of the OBJ file that should be loaded
`faceNormals`	`boolean`	When set to `true`, any existing or computed vertex normals are discarded and *face normals* will instead be used during rendering. This gives the rendered object a faceted appearance. (Default: `false`)
`maxSmoothAngle`	`float`	When specified, Mitsuba will discard all vertex normals in the input mesh and rebuild them in a way that is sensitive to the presence of creases and corners. For more details on this parameter, see below. Disabled by default.
`flipNormals`	`boolean`	Optional flag to flip all normals. (Default: `false`, i.e. the normals are left unchanged).
`flipTexCoords`	`boolean`	Treat the vertical component of the texture as inverted? Most OBJ files use this convention. (Default: `true`, i.e. flip them to get the correct coordinates).
`toWorld`	`transform` or `animation`	Specifies an optional linear object-to-world transformation. (Default: none (i.e. object space = world space))
`collapse`	`boolean`	Collapse all contained meshes into a single object (Default: `false`)

(a) An example scene with both geometry and materials imported using the Wavefront OBJ mesh loader (Neu Rungholt model courtesy of `kescha`, converted from Minecraft to OBJ by Morgan McGuire)

This plugin implements a simple loader for Wavefront OBJ files. It handles meshes containing

triangles and quadrilaterals, and it also imports vertex normals and texture coordinates.

Loading an ordinary OBJ file is as simple as writing:

```
<shape type="obj">
    <string name="filename" value="myShape.obj"/>
</shape>
```

Material import: When the OBJ file references a Wavefront material description (a `.mtl` file), Mitsuba attempts to reproduce the material within and associate it with the shape. This is restricted to fairly basic materials and textures, hence in most cases it will be preferable to override this behavior by specifying an explicit Mitsuba BSDF that should be used instead. This can be done by passing it as a child argument, e.g.

```
<shape type="obj">
    <string name="filename" value="myShape.obj"/>
    <bsdf type="roughplastic">
        <rgb name="diffuseReflectance" value="0.2, 0.6, 0.3"/>
    </bsdf>
</shape>
```

The `mtl` material attributes that are automatically handled by Mitsuba include:

- Diffuse and glossy materials (optionally textured)
- Smooth glass and metal
- Textured transparency
- Bump maps

In some cases, OBJ files contain *multiple* objects with different associated materials. In this case, the materials can be overwritten individually, by specifying the corresponding names. For instance, if the OBJ file contains two materials named `Glass` and `Water`, these can be overwritten as follows

```
<shape type="obj">
    <string name="filename" value="myShape.obj"/>
    <bsdf name="Glass" type="dielectric">
        <float name="intIOR" value="1.5"/>
    </bsdf>
    <bsdf name="Water" type="dielectric">
        <float name="intIOR" value="1.333"/>
    </bsdf>
</shape>
```

The `maxSmoothAngle` parameter: When given a mesh without vertex normals, Mitsuba will by default create a smoothly varying normal field over the entire shape. This can produce undesirable output when the input mesh contains regions that are intentionally not smooth (i.e. corners, creases). Meshes that do include vertex normals sometimes incorrectly interpolate normals over such regions, leading to much the same problem.

The `maxSmoothAngle` parameter can be issued to force inspection of the dihedral angle associated with each edge in the input mesh and disable normal interpolation locally where this angle exceeds

a certain threshold value. A reasonable value might be something like `30` (degrees). The underlying analysis is somewhat costly and hence this parameter should only be used when it is actually needed (i.e. when the mesh contains creases or edges and does not come with valid vertex normals).

Remarks:

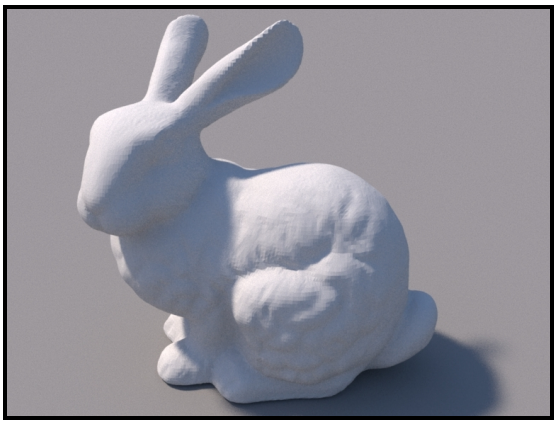
- The plugin currently only supports loading meshes constructed from triangles and quadrilaterals.
- Importing geometry via OBJ files should only be used as an absolutely last resort. Due to inherent limitations of this format, the files tend to be unreasonably large, and parsing them requires significant amounts of memory and processing power. What's worse is that the internally stored data is often truncated, causing a loss of precision.

  If possible, use the `ply` or `serialized` plugins instead. For convenience, it is also possible to convert legacy OBJ files into `.serialized` files using the `mtsimport` utility. Using the resulting output will significantly accelerate the scene loading time.

8.1.7. PLY (Stanford Triangle Format) mesh loader (`ply`)

Parameter	Type	Description
`filename`	`string`	Filename of the PLY file that should be loaded
`faceNormals`	`boolean`	When set to `true`, Mitsuba will use face normals when rendering the object, which will give it a faceted apperance. (Default: `false`)
`maxSmoothAngle`	`float`	When specified, Mitsuba will discard all vertex normals in the input mesh and rebuild them in a way that is sensitive to the presence of creases and corners. For more details on this parameter, see page 42. Disabled by default.
`flipNormals`	`boolean`	Optional flag to flip all normals. (Default: `false`, i.e. the normals are left unchanged).
`toWorld`	`transform` or `animation`	Specifies an optional linear object-to-world transformation. (Default: none (i.e. object space = world space))
`srgb`	`boolean`	When set to `true`, any vertex colors will be interpreted as sRGB, instead of linear RGB (Default: `true`).

(a) The PLY plugin is useful for loading large geometry. (Dragon statue courtesy of XYZ RGB)



(b) The Stanford bunny loaded with `faceNormals=true`. Note the faceted appearance.

This plugin implements a fast loader for the Stanford PLY format (both the ASCII and binary format). It is based on the `libply` library by Ares Lagae (`http://people.cs.kuleuven.be/~ares.lagae/libply`). The current plugin implementation supports triangle meshes with optional UV coordinates, vertex normals, and vertex colors.

When loading meshes that contain vertex colors, note that they need to be explicitly referenced in a BSDF using a special texture named `vertexcolors`.

8.1.8. Serialized mesh loader (`serialized`)

Parameter	Type	Description
`filename`	`string`	Filename of the geometry file that should be loaded
`shapeIndex`	`integer`	A `.serialized` file may contain several separate meshes. This parameter specifies which one should be loaded. (Default: `0`, i.e. the first one)
`faceNormals`	`boolean`	When set to `true`, any existing or computed vertex normals are discarded and *face normals* will instead be used during rendering. This gives the rendered object a faceted appearance. (Default: `false`)
`maxSmoothAngle`	`float`	When specified, Mitsuba will discard all vertex normals in the input mesh and rebuild them in a way that is sensitive to the presence of creases and corners. For more details on this parameter, see page 42. Disabled by default.
`flipNormals`	`boolean`	Optional flag to flip all normals. (Default: `false`, i.e. the normals are left unchanged).
`toWorld`	`transform` or `animation`	Specifies an optional linear object-to-world transformation. (Default: none (i.e. object space = world space))

The serialized mesh format represents the most space and time-efficient way of getting geometry information into Mitsuba. It stores indexed triangle meshes in a lossless gzip-based encoding that (after decompression) nicely matches up with the internally used data structures. Loading such files is considerably faster than the `ply` plugin and orders of magnitude faster than the `obj` plugin.

Format description: The `serialized` file format uses the little endian encoding, hence all fields below should be interpreted accordingly. The contents are structured as follows:

Type	Content
`uint16`	File format identifier: `0x041C`
`uint16`	File version identifier. Currently set to `0x0004`
	From this point on, the stream is compressed by the `DEFLATE` algorithm. The used encoding is that of the `zlib` library.
`uint32`	An 32-bit integer whose bits can be used to specify the following flags: **`0x0001`** The mesh data includes per-vertex normals **`0x0002`** The mesh data includes texture coordinates **`0x0008`** The mesh data includes vertex colors **`0x0010`** Use face normals instead of smothly interpolated vertex normals. Equivalent to specifying `faceNormals=true` to the plugin. **`0x1000`** The subsequent content is represented in single precision **`0x2000`** The subsequent content is represented in double precision
`string`	A null-terminated string (utf-8), which denotes the name of the shape.

uint64	Number of vertices in the mesh
uint64	Number of triangles in the mesh
array	Array of all vertex positions (X, Y, Z, X, Y, Z, ...) specified in binary single or double precision format (as denoted by the flags)
array	Array of all vertex normal directions (X, Y, Z, X, Y, Z, ...) specified in binary single or double precision format. When the mesh has no vertex normals, this field is omitted.
array	Array of all vertex texture coordinates (U, V, U, V, ...) specified in binary single or double precision format. When the mesh has no texture coordinates, this field is omitted.
array	Array of all vertex colors (R, G, B, R, G, B, ...) specified in binary single or double precision format. When the mesh has no vertex colors, this field is omitted.
array	Indexed triangle data (`[i1, i2, i3]`, `[i1, i2, i3]`, ..) specified in `uint32` or in `uint64` format (the latter is used when the number of vertices exceeds `0xFFFFFFFF`).

Multiple shapes: It is possible to store multiple meshes in a single `.serialized` file. This is done by simply concatenating their data streams, where every one is structured according to the above description. Hence, after each mesh, the stream briefly reverts back to an uncompressed format, followed by an uncompressed header, and so on. This is neccessary for efficient read access to arbitrary sub-meshes.

End-of-file dictionary: In addition to the previous table, a `.serialized` file also concludes with a brief summary at the end of the file, which specifies the starting position of each sub-mesh:

Type	Content
uint64	File offset of the first mesh (in bytes)—this is always zero.
uint64	File offset of the second mesh
…	…
uint64	File offset of the last sub-shape
uint32	Total number of meshes in the `.serialized` file

8.1.9. Shape group for geometry instancing (shapegroup)

Parameter	Type	Description
(Nested plugin)	shape	One or more shapes that should be made available for geometry instancing

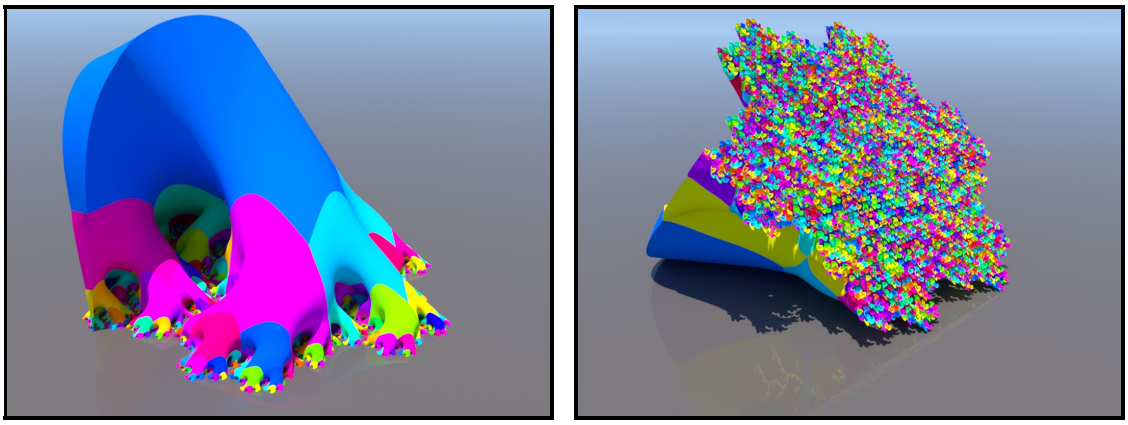
This plugin implements a container for shapes that should be made available for geometry instancing. Any shapes placed in a `shapegroup` will not be visible on their own—instead, the renderer will precompute ray intersection acceleration data structures so that they can efficiently be referenced many times using the `instance` plugin. This is useful for rendering things like forests, where only a few distinct types of trees have to be kept in memory. An example is given below:

```
<!-- Declare a named shape group containing two objects -->
<shape type="shapegroup" id="myShapeGroup">
    <shape type="ply">
        <string name="filename" value="data.ply"/>
        <bsdf type="roughconductor"/>
    </shape>
    <shape type="sphere">
        <transform name="toWorld">
            <scale value="5"/>
            <translate y="20"/>
        </transform>
        <bsdf type="diffuse"/>
    </shape>
</shape>

<!-- Instantiate the shape group without any kind of transformation -->
<shape type="instance">
    <ref id="myShapeGroup"/>
</shape>

<!-- Create instance of the shape group, but rotated, scaled, and translated -->
<shape type="instance">
    <ref id="myShapeGroup"/>
    <transform name="toWorld">
        <rotate x="1" angle="45"/>
        <scale value="1.5"/>
        <translate z="10"/>
    </transform>
</shape>
```

Listing 9: An example of geometry instancing

8.1.10. Geometry instance (`instance`)

Parameter	Type	Description
(Nested plugin)	`shapegroup`	A reference to a shape group that should be instantiated
`toWorld`	`transform` or `animation`	Specifies an optional linear instance-to-world transformation. (Default: none (i.e. instance space = world space))



(a) Surface viewed from the top (b) Surface viewed from the bottom

Figure 4: A visualization of a fractal surface by Irving and Segerman. (a 2D Gospel curve developed up to level 5 along the third dimension). This scene makes use of instancing to replicate similar structures to cheaply render a shape that effectively consists of several hundred millions of triangles.

This plugin implements a geometry instance used to efficiently replicate geometry many times. For details on how to create instances, refer to the `shapegroup` plugin.

Remarks:

- Note that it is *not* possible to assign a different material to each instance — the material assignment specified within the shape group is the one that matters.
- Shape groups cannot be used to replicate shapes with attached emitters, sensors, or subsurface scattering models.

8.1.11. Hair intersection shape (`hair`)

Parameter	Type	Description
`filename`	`string`	Filename of the hair data file that should be loaded
`radius`	`float`	Radius of the hair segments in world-space units (Default: 0.025, which assumes that the scene is modeled in millimeters.).
`angleThreshold`	`float`	For performance reasons, the plugin will merge adjacent hair segments when the angle of their tangent directions is below than this value (in degrees). (Default: 1).
`reduction`	`float`	When the reduction ratio is set to a value between zero and one, the hair plugin stochastically culls this portion of the input data (where 1 corresponds to removing all hairs). To approximately preserve the appearance in renderings, the hair radius is enlarged (see Cook et al. [6]). This parameter is convenient for fast previews. (Default: 0, i.e. all geometry is rendered)
`toWorld`	`transform`	Specifies an optional linear object-to-world transformation. Note that non-uniform scales are not permitted! (Default: none, i.e. object space = world space)

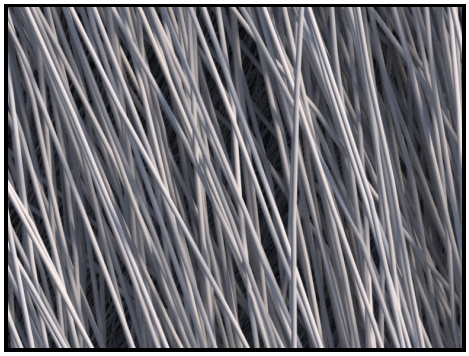


Figure 5: A close-up of the hair shape rendered with a diffuse scattering model (an actual hair scattering model will be needed for realistic apperance)

The plugin implements a space-efficient acceleration structure for hairs made from many straight cylindrical hair segments with miter joints. The underlying idea is that intersections with straight cylindrical hairs can be found quite efficiently, and curved hairs are easily approximated using a series of such segments.

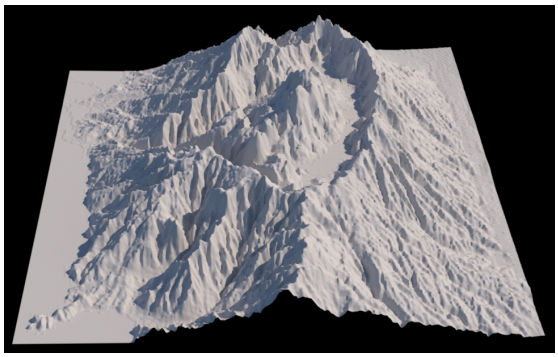
The plugin supports two different input formats: a simple (but not particularly efficient) ASCII format containing the coordinates of a hair vertex on every line. An empty line marks the beginning of a new hair. The following snippet is an example of this format:

```
.....
-18.5498 -21.7669 22.8138
-18.6358 -21.3581 22.9262
-18.7359 -20.9494 23.0256

-30.6367 -21.8369 6.78397
-30.7289 -21.4145 6.76688
-30.8226 -20.9933 6.73948
.....
```

There is also a binary format, which starts with the identifier "`BINARY_HAIR`" (11 bytes), followed by the number of vertices as a 32-bit little endian integer. The remainder of the file consists of the vertex positions stored as single-precision XYZ coordinates (again in little-endian byte ordering). To mark the beginning of a new hair strand, a single $+\infty$ floating point value can be inserted between the vertex data.

8.1.12. Height field intersection shape (heightfield)

Parameter	Type	Description
shadingNormals	boolean	Use linearly interpolated shading normals over the height field as opposed to discontinuous normals from the underlying bilinear patches? (Default: true, i.e. interpolate smoothly varying normals)
flipNormals	boolean	Optional flag to flip all normals. (Default: false, i.e. the normals are left unchanged).
toWorld	transform	Specifies an optional linear object-to-world transformation. (Default: none, i.e. object space = world space)
width, height	integer	When the nested texture is procedural (see below), this parameter specifies the resolution at which it should be rasterized to create a height field made of bilinear patches.
scale	float	Scale factor that is applied to the height field values (Default: No scaling, i.e. 1)
filename	string	File name of an image file containing height field values. Alternatively, a nested texture can be provided (see below).
(Nested plugin)	texture	A nested texture that specifies the height field values. This could be a bitmap-backed texture or one that is procedurally defined. In the latter case, it will be rasterized using the resolution specified by the width and height arguments.



(a) Heigh field rendering of a mountain, see Listing 10

This plugin implements an efficient height field intersection shape, i.e. a two-dimensional plane that is vertically displaced using height values loaded from a texture. Internally, the height field is represented as a min-max mipmap [44], allowing cheap storage and efficient ray intersection queries. It is generally preferable to represent height fields using this specialized plugin rather than converting them into triangle meshes.

```
<shape type="heightfield">
    <string name="filename" value="mountain_profile.exr"/>
    <float name="scale" value="0.5"/>
</shape>
```

Listing 10: Declaring a height field from a monochromatic scaled bitmap texture

8.2. Surface scattering models

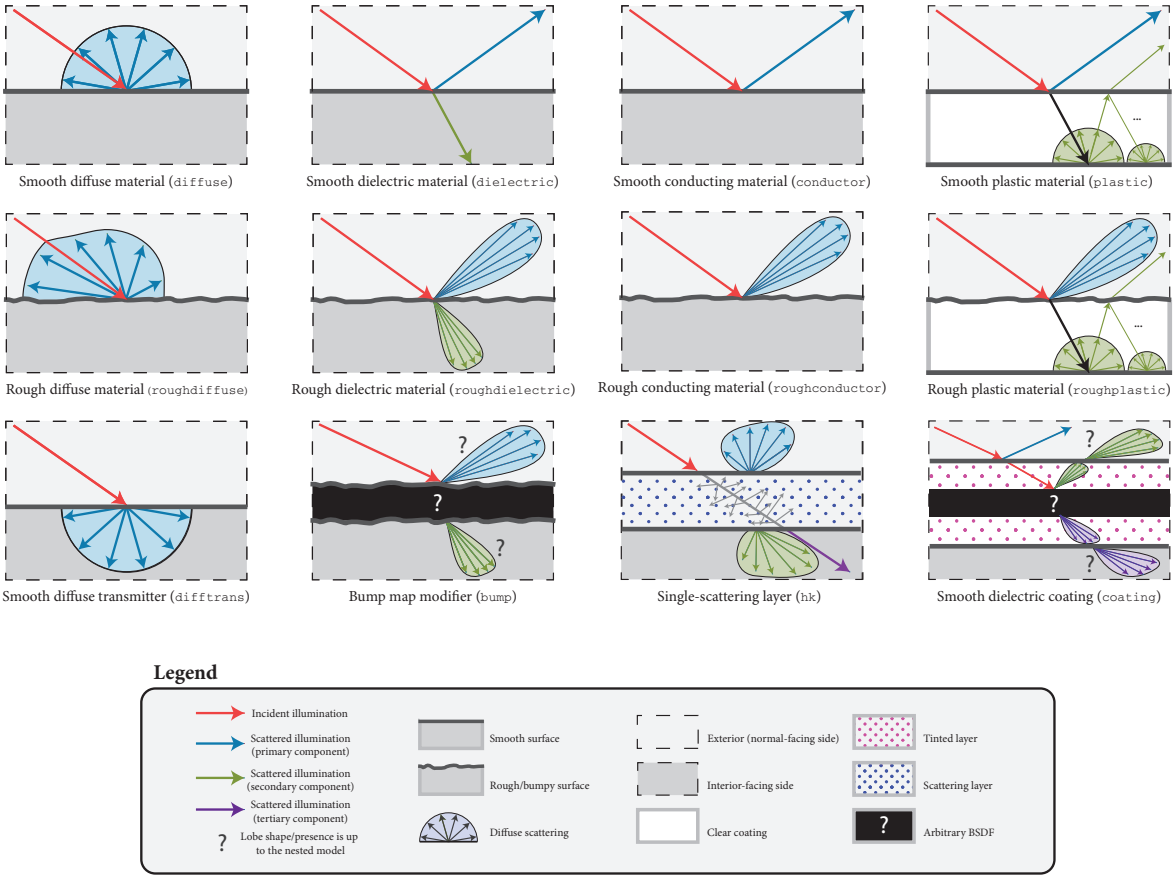


Figure 6: Schematic overview of the most important surface scattering models in Mitsuba (shown in the style of Weidlich and Wilkie [52]). The arrows indicate possible outcomes of an interaction with a surface that has the respective model applied to it.

Surface scattering models describe the manner in which light interacts with surfaces in the scene. They conveniently summarize the mesoscopic scattering processes that take place within the material and cause it to look the way it does. This represents one central component of the material system in Mitsuba—another part of the renderer concerns itself with what happens *in between* surface interactions. For more information on this aspect, please refer to Sections 8.5 and 8.4. This section presents an overview of all surface scattering models that are supported, along with their parameters.

BSDFs

To achieve realistic results, Mitsuba comes with a library of both general-purpose surface scattering models (smooth or rough glass, metal, plastic, etc.) and specializations to particular materials (woven cloth, masks, etc.). Some model plugins fit neither category and can best be described as *modifiers* that are applied on top of one or more scattering models.

Throughout the documentation and within the scene description language, the word *BSDF* is used synonymously with the term "surface scattering model". This is an abbreviation for *Bidirectional Scat-*

tering Distribution Function, a more precise technical term.

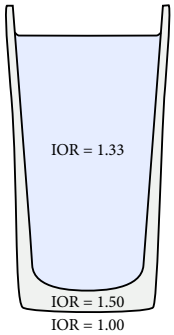
In Mitsuba, BSDFs are assigned to *shapes*, which describe the visible surfaces in the scene. In the scene description language, this assignment can either be performed by nesting BSDFs within shapes, or they can be named and then later referenced by their name. The following fragment shows an example of both kinds of usages:

```
<scene version="0.5.0">
    <!-- Creating a named BSDF for later use -->
    <bsdf type=".. BSDF type .." id="myNamedMaterial">
        <!-- BSDF parameters go here -->
    </bsdf>

    <shape type="sphere">
        <!-- Example of referencing a named material -->
        <ref id="myNamedMaterial"/>
    </shape>

    <shape type="sphere">
        <!-- Example of instantiating an unnamed material -->
        <bsdf type=".. BSDF type ..">
            <!-- BSDF parameters go here -->
        </bsdf>
    </shape>
</scene>
```

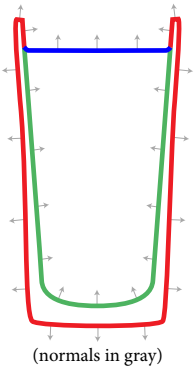
It is generally more economical to use named BSDFs when they are used in several places, since this reduces Mitsuba's internal memory usage.

Correctness considerations

A vital consideration when modeling a scene in a physically-based rendering system is that the used materials do not violate physical properties, and that their arrangement is meaningful. For instance,



(a) Slice through a glass filled with water



(b) Description using surfaces in Mitsuba

Surface	Interior IOR	Exterior IOR
	1.50	1.00
	1.50	1.33
	1.33	1.00

(c) Detailed IOR transitions

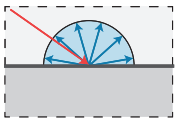
Figure 7: Some of the scattering models in Mitsuba need to know the indices of refraction on the exterior and interior-facing side of a surface. It is therefore important to decompose the mesh into meaningful separate surfaces corresponding to each index of refraction change. The example here shows such a decomposition for a water-filled Glass.

imagine having designed an architectural interior scene that looks good except for a white desk that seems a bit too dark. A closer inspection reveals that it uses a Lambertian material with a diffuse reflectance of 0.9.

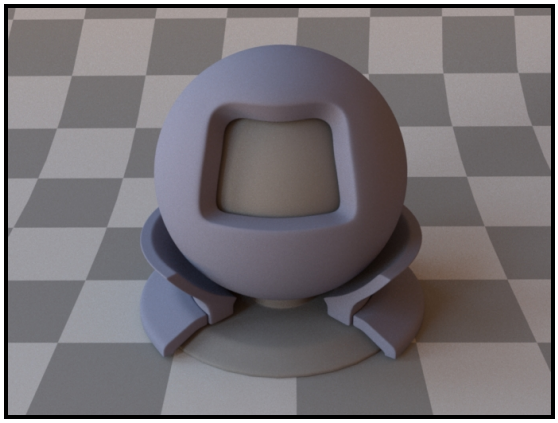
In many rendering systems, it would be feasible to increase the reflectance value above 1.0 in such a situation. But in Mitsuba, even a small surface that reflects a little more light than it receives will likely break the available rendering algorithms, or cause them to produce otherwise unpredictable results. In fact, the right solution in this case would be to switch to a different the lighting setup that causes more illumination to be received by the desk and then *reduce* the material's reflectance—after all, it is quite unlikely that one could find a real-world desk that reflects 90% of all incident light.

As another example of the necessity for a meaningful material description, consider the glass model illustrated in Figure 7. Here, careful thinking is needed to decompose the object into boundaries that mark index of refraction-changes. If this is done incorrectly and a beam of light can potentially pass through a sequence of incompatible index of refraction changes (e.g. 1.00 → 1.33 followed by 1.50 → 1.33), the output is undefined and will quite likely even contain inaccuracies in parts of the scene that are far away from the glass.

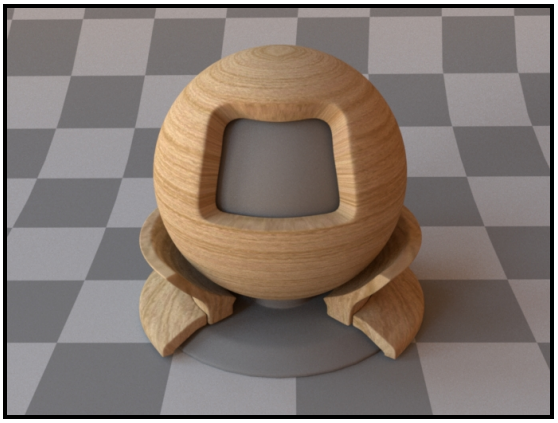
8.2.1. Smooth diffuse material (`diffuse`)



Parameter	Type	Description
`reflectance`	`spectrum` or `texture`	Specifies the diffuse albedo of the material (Default: 0.5)



(a) Homogeneous reflectance, see Listing 11



(b) Textured reflectance, see Listing 12

The smooth diffuse material (also referred to as "Lambertian") represents an ideally diffuse material with a user-specified amount of reflectance. Any received illumination is scattered so that the surface looks the same independently of the direction of observation.

Apart from a homogeneous reflectance value, the plugin can also accept a nested or referenced texture map to be used as the source of reflectance information, which is then mapped onto the shape based on its UV parameterization. When no parameters are specified, the model uses the default of 50% reflectance.

Note that this material is one-sided—that is, observed from the back side, it will be completely black. If this is undesirable, consider using the `twosided` BRDF adapter plugin.

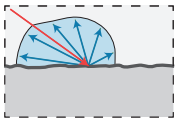
```
<bsdf type="diffuse">
    <srgb name="reflectance" value="#6d7185"/>
</bsdf>
```

Listing 11: A diffuse material, whose reflectance is specified as an sRGB color

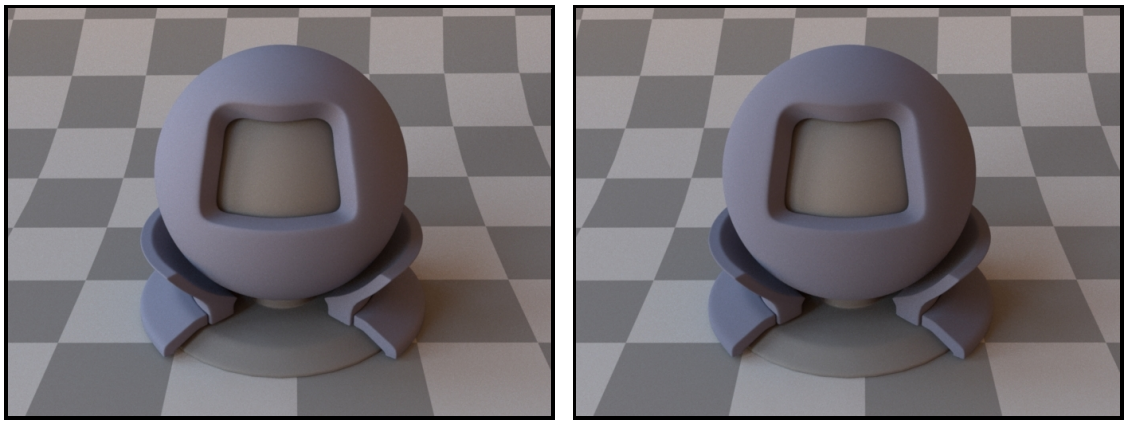
```
<bsdf type="diffuse">
    <texture type="bitmap" name="reflectance">
        <string name="filename" value="wood.jpg"/>
    </texture>
</bsdf>
```

Listing 12: A diffuse material with a texture map

8.2.2. Rough diffuse material (`roughdiffuse`)



Parameter	Type	Description
`reflectance`	`spectrum` or `texture`	Specifies the diffuse albedo of the material. (Default: 0.5)
`alpha`	`spectrum` or `texture`	Specifies the roughness of the unresolved surface microgeometry using the *root mean square* (RMS) slope of the microfacets. (Default: 0.2)
`useFastApprox`	`boolean`	This parameter selects between the full version of the model or a fast approximation that still retains most qualitative features. (Default: `false`, i.e. use the high-quality version)



(a) Smooth diffuse surface ($\alpha = 0$) (b) Very rough diffuse surface ($\alpha = 0.7$)

Figure 8: The effect of switching from smooth to rough diffuse scattering is fairly subtle on this model—generally, there will be higher reflectance at grazing angles, as well as an overall reduced contrast.

This reflectance model describes the interaction of light with a *rough* diffuse material, such as plaster, sand, clay, or concrete, or "powdery" surfaces. The underlying theory was developed by Oren and Nayar [35], who model the microscopic surface structure as unresolved planar facets arranged in V-shaped grooves, where each facet is an ideal diffuse reflector. The model takes into account shadowing, masking, as well as interreflections between the facets.

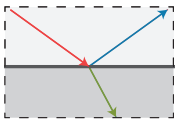
Since the original publication, this approach has been shown to be a good match for many real-world materials, particularly compared to Lambertian scattering, which does not take surface roughness into account.

The implementation in Mitsuba uses a surface roughness parameter α that is slightly different from the slope-area variance in the original 1994 paper. The reason for this change is to make the parameter α portable across different models (i.e. `roughdielectric`, `roughplastic`, `roughconductor`).

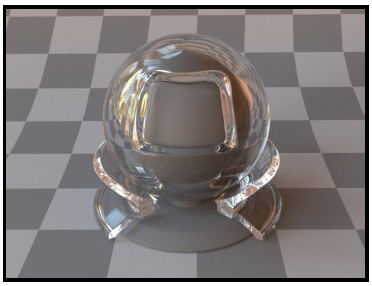
To get an intuition about the effect of the parameter α, consider the following approximate classification: a value of $\alpha = 0.001 - 0.01$ corresponds to a material with slight imperfections on an otherwise smooth surface (for such small values, the model will behave identically to `diffuse`), $\alpha = 0.1$ is relatively rough, and $\alpha = 0.3 - 0.7$ is *extremely* rough (e.g. an etched or ground surface).

Note that this material is one-sided—that is, observed from the back side, it will be completely black. If this is undesirable, consider using the `twosided` BRDF adapter plugin.

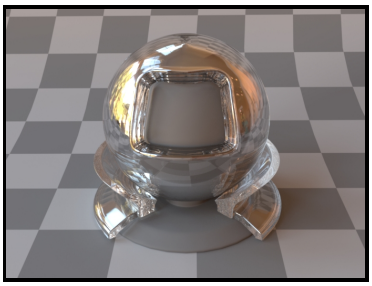
8.2.3. Smooth dielectric material (`dielectric`)



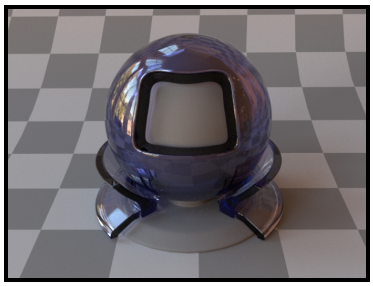
Parameter	Type	Description
`intIOR`	`float` or `string`	Interior index of refraction specified numerically or using a known material name. (Default: `bk7` / 1.5046)
`extIOR`	`float` or `string`	Exterior index of refraction specified numerically or using a known material name. (Default: `air` / 1.000277)
`specular↙ Reflectance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)
`specular↙ Transmittance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular transmission component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)



(a) Air↔Water (IOR: 1.33) interface. See Listing 13.



(b) Air↔Diamond (IOR: 2.419)



(c) Air↔Glass (IOR: 1.504) interface with absorption. See Listing 14.

This plugin models an interface between two dielectric materials having mismatched indices of refraction (for instance, water and air). Exterior and interior IOR values can be specified independently, where "exterior" refers to the side that contains the surface normal. When no parameters are given, the plugin activates the defaults, which describe a borosilicate glass BK7/air interface.

In this model, the microscopic structure of the surface is assumed to be perfectly smooth, resulting in a degenerate[10] BSDF described by a Dirac delta distribution. For a similar model that instead describes a rough surface microstructure, take a look at the `roughdielectric` plugin.

```
<shape type="...">
    <bsdf type="dielectric">
        <string name="intIOR" value="water"/>
        <string name="extIOR" value="air"/>
    </bsdf>
<shape>
```

Listing 13: A simple air-to-water interface

When using this model, it is crucial that the scene contains meaningful and mutually compatible indices of refraction changes—see Figure 7 for a description of what this entails.

In many cases, we will want to additionally describe the *medium* within a dielectric material. This

[10] Meaning that for any given incoming ray of light, the model always scatters into a discrete set of directions, as opposed to a continuum.

requires the use of a rendering technique that is aware of media (e.g. the volumetric path tracer). An example of how one might describe a slightly absorbing piece of glass is shown below:

```
<shape type="...">
    <bsdf type="dielectric">
        <float name="intIOR" value="1.504"/>
        <float name="extIOR" value="1.0"/>
    </bsdf>

    <medium type="homogeneous" name="interior">
        <rgb name="sigmaS" value="0, 0, 0"/>
        <rgb name="sigmaA" value="4, 4, 2"/>
    </medium>
<shape>
```

Listing 14: A glass material with absorption (based on the Beer-Lambert law). This material can only be used by an integrator that is aware of participating media.

Name	Value	Name	Value
vacuum	1.0	bromine	1.661
helium	1.00004	water ice	1.31
hydrogen	1.00013	fused quartz	1.458
air	1.00028	pyrex	1.470
carbon dioxide	1.00045	acrylic glass	1.49
water	1.3330	polypropylene	1.49
acetone	1.36	bk7	1.5046
ethanol	1.361	sodium chloride	1.544
carbon tetrachloride	1.461	amber	1.55
glycerol	1.4729	pet	1.575
benzene	1.501	diamond	2.419
silicone oil	1.52045		

Table 3: This table lists all supported material names along with along with their associated index of refraction at standard conditions. These material names can be used with the plugins `dielectric`, `roughdielectric`, `plastic`, `roughplastic`, as well as `coating`.

Remarks:

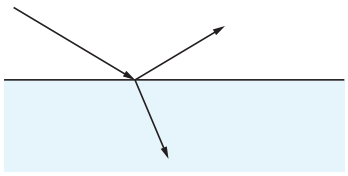
- Dispersion is currently unsupported but will be enabled in a future release.

8.2.4. Thin dielectric material (`thindielectric`)

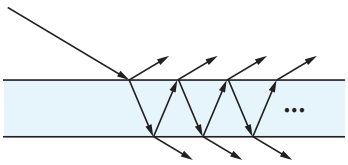
Parameter	Type	Description
`intIOR`	`float` or `string`	Interior index of refraction specified numerically or using a known material name. (Default: `bk7` / 1.5046)
`extIOR`	`float` or `string`	Exterior index of refraction specified numerically or using a known material name. (Default: `air` / 1.000277)
`specular↙ Reflectance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)
`specular↙ Transmittance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular transmission component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)

This plugin models a *thin* dielectric material that is embedded inside another dielectric—for instance, glass surrounded by air. The interior of the material is assumed to be so thin that its effect on transmitted rays is negligible, Hence, light exits such a material without any form of angular deflection (though there is still specular reflection).

This model should be used for things like glass windows that were modeled using only a single sheet of triangles or quads. On the other hand, when the window consists of proper closed geometry, `dielectric` is the right choice. This is illustrated below:



(a) The `dielectric` plugin models a single transition from one index of refraction to another



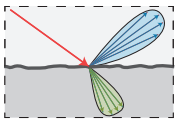
(b) The `thindielectric` plugin models a pair of interfaces causing a transient index of refraction change

(c) Windows modeled using a single sheet of geometry are the most frequent application of this BSDF

Figure 9: An illustration of the difference between the `dielectric` and `thindielectric` plugins

The implementation correctly accounts for multiple internal reflections inside the thin dielectric at *no significant extra cost*, i.e. paths of the type R, TRT, TR^3T, .. for reflection and TT, TR^2, TR^4T, .. for refraction, where T and R denote individual reflection and refraction events, respectively.

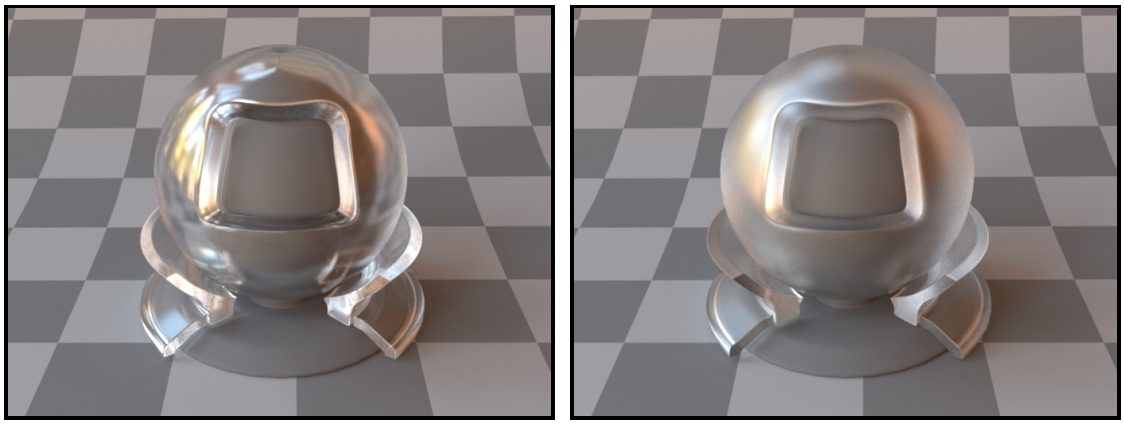
8.2.5. Rough dielectric material (`roughdielectric`)



Parameter	Type	Description
`distribution`	`string`	Specifies the type of microfacet normal distribution used to model the surface roughness. (i) `beckmann`: Physically-based distribution derived from Gaussian random surfaces. This is the default. (ii) `ggx`: New distribution proposed by Walter et al. [48], which is meant to better handle the long tails observed in measurements of ground surfaces. Renderings with this distribution may converge slowly. (iii) `phong`: Classical $\cos^p\theta$ distribution. Due to the underlying microfacet theory, the use of this distribution here leads to more realistic behavior than the separately available phong plugin. (iv) `as`: Anisotropic Phong-style microfacet distribution proposed by Ashikhmin and Shirley [1].
`alpha`	`float` or `texture`	Specifies the roughness of the unresolved surface microgeometry. When the Beckmann distribution is used, this parameter is equal to the *root mean square* (RMS) slope of the microfacets. This parameter is only valid when `distribution=beckmann/phong/ggx`. (Default: 0.1).
`alphaU, alphaV`	`float` or `texture`	Specifies the anisotropic roughness values along the tangent and bitangent directions. These parameter are only valid when `distribution=as`. (Default: 0.1).
`intIOR`	`float` or `string`	Interior index of refraction specified numerically or using a known material name. (Default: `bk7` / 1.5046)
`extIOR`	`float` or `string`	Exterior index of refraction specified numerically or using a known material name. (Default: `air` / 1.000277)
`specular↙Reflectance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)
`specular↙Transmittance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular transmission component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)

This plugin implements a realistic microfacet scattering model for rendering rough interfaces between dielectric materials, such as a transition from air to ground glass. Microfacet theory describes rough surfaces as an arrangement of unresolved and ideally specular facets, whose normal directions are given by a specially chosen *microfacet distribution*. By accounting for shadowing and masking effects between these facets, it is possible to reproduce the important off-specular reflections peaks observed in real-world measurements of such materials.

This plugin is essentially the "roughened" equivalent of the (smooth) plugin `dielectric`. For very



(a) Anti-glare glass (Beckmann, $\alpha = 0.02$) (b) Rough glass (Beckmann, $\alpha = 0.1$)

low values of α, the two will be identical, though scenes using this plugin will take longer to render due to the additional computational burden of tracking surface roughness.

The implementation is based on the paper "Microfacet Models for Refraction through Rough Surfaces" by Walter et al. [48]. It supports several different types of microfacet distributions and has a texturable roughness parameter. Exterior and interior IOR values can be specified independently, where "exterior" refers to the side that contains the surface normal. Similar to the `dielectric` plugin, IOR values can either be specified numerically, or based on a list of known materials (see Table 3 for an overview). When no parameters are given, the plugin activates the default settings, which describe a borosilicate glass BK7/air interface with a light amount of roughness modeled using a Beckmann distribution.

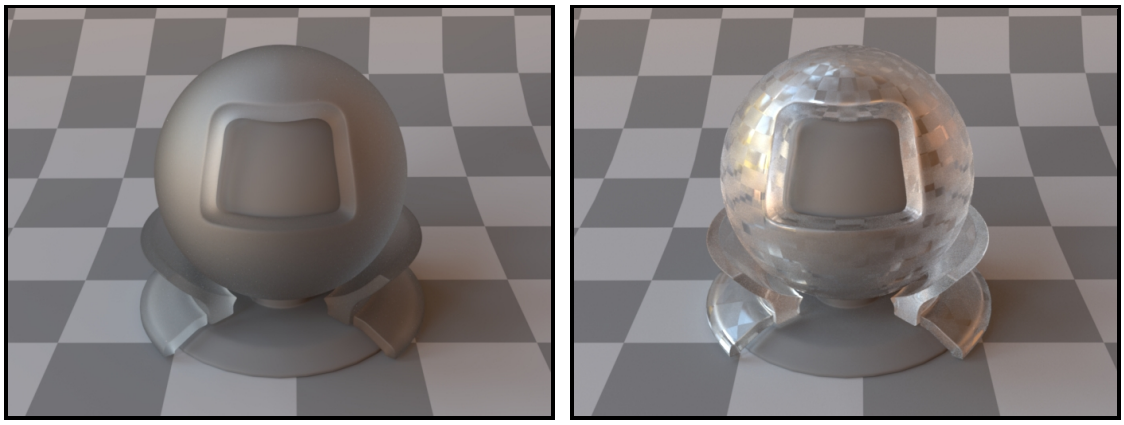
To get an intuition about the effect of the surface roughness parameter α, consider the following approximate classification: a value of $\alpha = 0.001 - 0.01$ corresponds to a material with slight imperfections on an otherwise smooth surface finish, $\alpha = 0.1$ is relatively rough, and $\alpha = 0.3 - 0.7$ is *extremely* rough (e.g. an etched or ground finish).

Please note that when using this plugin, it is crucial that the scene contains meaningful and mutually compatible index of refraction changes—see Figure 7 for an example of what this entails. Also, note that the importance sampling implementation of this model is close, but not always a perfect a perfect match to the underlying scattering distribution, particularly for high roughness values and when the `ggx` microfacet distribution is used. Hence, such renderings may converge slowly.

Technical details

When rendering with the Ashikhmin-Shirley or Phong microfacet distributions, a conversion is used to turn the specified α roughness value into the exponents of these distributions. This is done in a way, such that the different distributions all produce a similar appearance for the same value of α.

The Ashikhmin-Shirley microfacet distribution allows the specification of two distinct roughness values along the tangent and bitangent directions. This can be used to provide a material with a "brushed" appearance. The alignment of the anisotropy will follow the UV parameterization of the underlying mesh in this case. This also means that such an anisotropic material cannot be applied to triangle meshes that are missing texture coordinates.



(a) Ground glass (GGX, α=0.304, Listing 15) (b) Textured roughness (Listing 16)

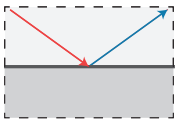
```
<bsdf type="roughdielectric">
    <string name="distribution" value="ggx"/>
    <float name="alpha" value="0.304"/>
    <string name="intIOR" value="bk7"/>
    <string name="extIOR" value="air"/>
</bsdf>
```

Listing 15: A material definition for ground glass

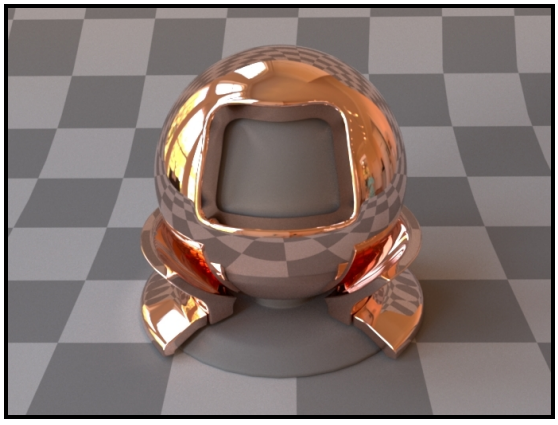
```
<bsdf type="roughdielectric">
    <string name="distribution" value="beckmann"/>
    <float name="intIOR" value="1.5046"/>
    <float name="extIOR" value="1.0"/>

    <texture name="alpha" type="bitmap">
        <string name="filename" value="roughness.exr"/>
    </texture>
</bsdf>
```

Listing 16: A texture can be attached to the roughness parameter

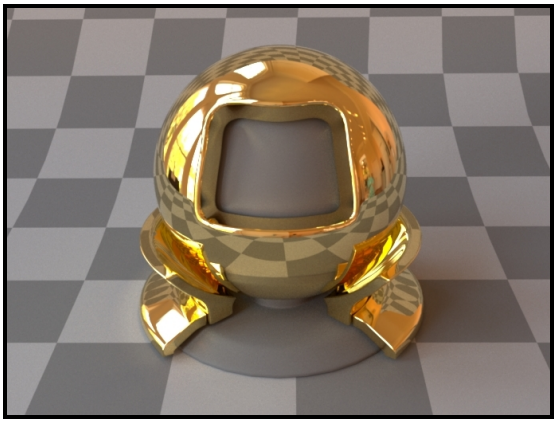
8.2.6. Smooth conductor (`conductor`)



Parameter	Type	Description
`material`	`string`	Name of a material preset, see Table 4. (Default: `Cu` / copper)
`eta, k`	`spectrum`	Real and imaginary components of the material's index of refraction (Default: based on the value of `material`)
`extEta`	`float` or `string`	Real-valued index of refraction of the surrounding dielectric, or a material name of a dielectric (Default: `air`)
`specular↙ Reflectance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)



(a) Measured copper material (the default), rendered using 30 spectral samples between 360 and 830nm



(b) Measured gold material (Listing 17)

This plugin implements a perfectly smooth interface to a conducting material, such as a metal. For a similar model that instead describes a rough surface microstructure, take a look at the separately available `roughconductor` plugin.

In contrast to dielectric materials, conductors do not transmit any light. Their index of refraction is complex-valued and tends to undergo considerable changes throughout the visible color spectrum.

To facilitate the tedious task of specifying spectrally-varying index of refraction information, Mitsuba ships with a set of measured data for several materials, where visible-spectrum information was publicly available[11].

Note that Table 4 also includes several popular optical coatings, which are not actually conductors. These materials can also be used with this plugin, though note that the plugin will ignore any refraction component that the actual material might have had. There is also a special material profile named `none`, which disables the computation of Fresnel reflectances and produces an idealized 100% reflecting mirror.

When using this plugin, you should ideally compile Mitsuba with support for spectral rendering to get the most accurate results. While it also works in RGB mode, the computations will be more

[11] These index of refraction values are identical to the data distributed with PBRT. They are originally from the Luxpop database (`www.luxpop.com`) and are based on data by Palik et al. [36] and measurements of atomic scattering factors made by the Center For X-Ray Optics (CXRO) at Berkeley and the Lawrence Livermore National Laboratory (LLNL).

approximate in nature. Also note that this material is one-sided—that is, observed from the back side, it will be completely black. If this is undesirable, consider using the `twosided` BRDF adapter plugin.

```
<shape type="...">
    <bsdf type="conductor">
        <string name="material" value="Au"/>
    </bsdf>
<shape>
```

Listing 17: A material configuration for a smooth conductor with measured gold data

It is also possible to load spectrally varying index of refraction data from two external files containing the real and imaginary components, respectively (see Section 6.1.4 for details on the file format):

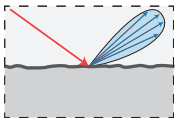
```
<shape type="...">
    <bsdf type="conductor">
        <spectrum name="eta" filename="conductorIOR.eta.spd"/>
        <spectrum name="k" filename="conductorIOR.k.spd"/>
    </bsdf>
<shape>
```

Listing 18: Rendering a smooth conductor with custom data

Preset(s)	Description	Preset(s)	Description
a-C	Amorphous carbon	Na_palik	Sodium
Ag	Silver	Nb, Nb_palik	Niobium
Al	Aluminium	Ni_palik	Nickel
AlAs, AlAs_palik	Cubic aluminium arsenide	Rh, Rh_palik	Rhodium
AlSb, AlSb_palik	Cubic aluminium antimonide	Se, Se_palik	Selenium
Au	Gold	SiC, SiC_palik	Hexagonal silicon carbide
Be, Be_palik	Polycrystalline beryllium	SnTe, SnTe_palik	Tin telluride
Cr	Chromium	Ta, Ta_palik	Tantalum
CsI, CsI_palik	Cubic caesium iodide	Te, Te_palik	Trigonal tellurium
Cu, Cu_palik	Copper	ThF4, ThF4_palik	Polycryst. thorium (IV) fluoride
Cu2O, Cu2O_palik	Copper (I) oxide	TiC, TiC_palik	Polycrystalline titanium carbide
CuO, CuO_palik	Copper (II) oxide	TiN, TiN_palik	Titanium nitride
d-C, d-C_palik	Cubic diamond	TiO2, TiO2_palik	Tetragonal titan. dioxide
Hg, Hg_palik	Mercury	VC, VC_palik	Vanadium carbide
HgTe, HgTe_palik	Mercury telluride	V_palik	Vanadium
Ir, Ir_palik	Iridium	VN, VN_palik	Vanadium nitride
K, K_palik	Polycrystalline potassium	W	Tungsten
Li, Li_palik	Lithium		
MgO, MgO_palik	Magnesium oxide		
Mo, Mo_palik	Molybdenum	none	No mat. profile (→ 100% reflecting mirror)

Table 4: This table lists all supported materials that can be passed into the `conductor` and `roughconductor` plugins. Note that some of them are not actually conductors—this is not a problem, they can be used regardless (though only the reflection component and no transmission will be simulated). In most cases, there are multiple entries for each material, which represent measurements by different authors.

8.2.7. Rough conductor material (roughconductor)

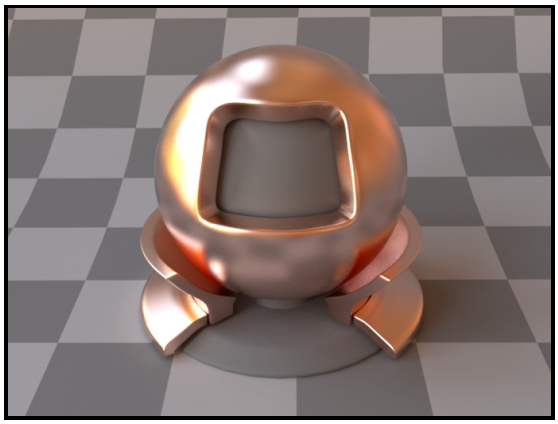


Parameter	Type	Description
distribution	string	Specifies the type of microfacet normal distribution used to model the surface roughness. (i) beckmann: Physically-based distribution derived from Gaussian random surfaces. This is the default. (ii) ggx: New distribution proposed by Walter et al. [48], which is meant to better handle the long tails observed in measurements of ground surfaces. Renderings with this distribution may converge slowly. (iii) phong: Classical $\cos^p\theta$ distribution. Due to the underlying microfacet theory, the use of this distribution here leads to more realistic behavior than the separately available phong plugin. (iv) as: Anisotropic Phong-style microfacet distribution proposed by Ashikhmin and Shirley [1].
alpha	float or texture	Specifies the roughness of the unresolved surface microgeometry. When the Beckmann distribution is used, this parameter is equal to the *root mean square* (RMS) slope of the microfacets. This parameter is only valid when distribution=beckmann/phong/ggx. (Default: 0.1).
alphaU, alphaV	float or texture	Specifies the anisotropic roughness values along the tangent and bitangent directions. These parameter are only valid when distribution=as. (Default: 0.1).
material	string	Name of a material preset, see Table 4. (Default: Cu / copper)
eta, k	spectrum	Real and imaginary components of the material's index of refraction (Default: based on the value of material)
extEta	float or string	Real-valued index of refraction of the surrounding dielectric, or a material name of a dielectric (Default: air)
specular↙ Reflectance	spectrum or texture	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)

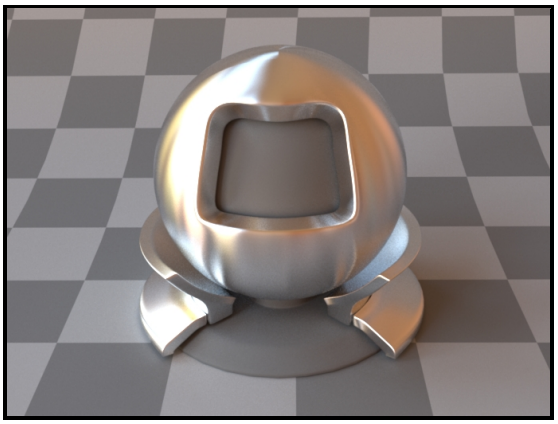
This plugin implements a realistic microfacet scattering model for rendering rough conducting materials, such as metals. It can be interpreted as a fancy version of the Cook-Torrance model and should be preferred over heuristic models like phong and ward when possible.

Microfacet theory describes rough surfaces as an arrangement of unresolved and ideally specular facets, whose normal directions are given by a specially chosen *microfacet distribution*. By accounting for shadowing and masking effects between these facets, it is possible to reproduce the important off-specular reflections peaks observed in real-world measurements of such materials.

This plugin is essentially the "roughened" equivalent of the (smooth) plugin conductor. For very low values of α, the two will be identical, though scenes using this plugin will take longer to render



(a) Rough copper (Beckmann, $\alpha = 0.1$)



(b) Vertically brushed aluminium (Ashikhmin-Shirley, $\alpha_u = 0.05$, $\alpha_v = 0.3$), see Listing 19

due to the additional computational burden of tracking surface roughness.

The implementation is based on the paper "Microfacet Models for Refraction through Rough Surfaces" by Walter et al. [48]. It supports several different types of microfacet distributions and has a texturable roughness parameter. To facilitate the tedious task of specifying spectrally-varying index of refraction information, this plugin can access a set of measured materials for which visible-spectrum information was publicly available (see Table 4 for the full list). There is also a special material profile named `none`, which disables the computation of Fresnel reflectances and produces an idealized 100% reflecting mirror.

When no parameters are given, the plugin activates the default settings, which describe copper with a light amount of roughness modeled using a Beckmann distribution.

To get an intuition about the effect of the surface roughness parameter α, consider the following approximate classification: a value of $\alpha = 0.001 - 0.01$ corresponds to a material with slight imperfections on an otherwise smooth surface finish, $\alpha = 0.1$ is relatively rough, and $\alpha = 0.3 - 0.7$ is *extremely* rough (e.g. an etched or ground finish). Values significantly above that are probably not too realistic.

```
<bsdf type="roughconductor">
    <string name="material" value="Al"/>
    <string name="distribution" value="as"/>
    <float name="alphaU" value="0.05"/>
    <float name="alphaV" value="0.3"/>
</bsdf>
```

Listing 19: A material definition for brushed aluminium

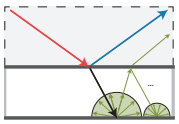
Technical details

When rendering with the Ashikhmin-Shirley or Phong microfacet distributions, a conversion is used to turn the specified α roughness value into the exponents of these distributions. This is done in a way, such that the different distributions all produce a similar appearance for the same value of α.

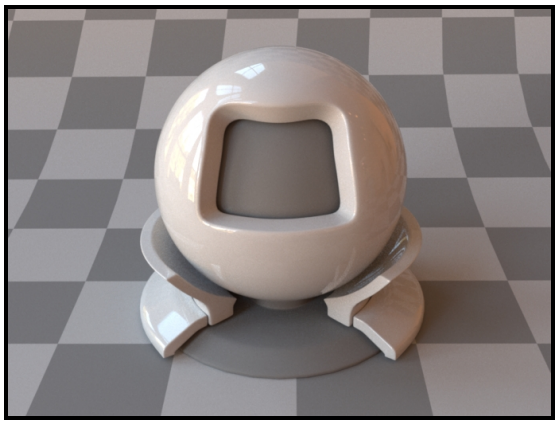
The Ashikhmin-Shirley microfacet distribution allows the specification of two distinct roughness values along the tangent and bitangent directions. This can be used to provide a material with a "brushed" appearance. The alignment of the anisotropy will follow the UV parameterization of the underlying mesh in this case. This also means that such an anisotropic material cannot be applied to triangle meshes that are missing texture coordinates.

When using this plugin, you should ideally compile Mitsuba with support for spectral rendering to get the most accurate results. While it also works in RGB mode, the computations will be more approximate in nature. Also note that this material is one-sided—that is, observed from the back side, it will be completely black. If this is undesirable, consider using the `twosided` BRDF adapter.

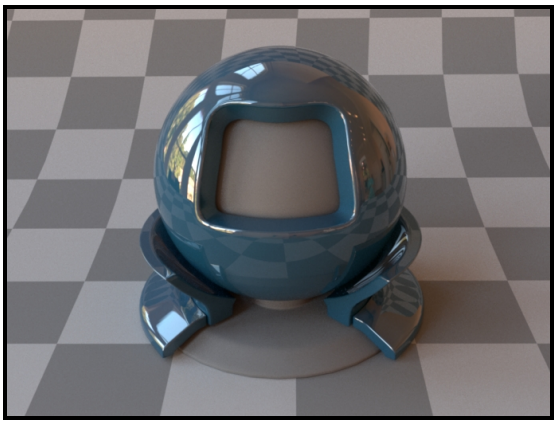
8.2.8. Smooth plastic material (plastic)



Parameter	Type	Description
intIOR	float or string	Interior index of refraction specified numerically or using a known material name. (Default: polypropylene / 1.49)
extIOR	float or string	Exterior index of refraction specified numerically or using a known material name. (Default: air / 1.000277)
specularReflectance	spectrum or texture	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)
diffuseReflectance	spectrum or texture	Optional factor used to modulate the diffuse reflection component (Default: 0.5)
nonlinear	boolean	Account for nonlinear color shifts due to internal scattering? See the main text for details. (Default: Don't account for them and preserve the texture colors, i.e. false)



(a) A rendering with the default parameters



(b) A rendering with custom parameters (Listing 20)

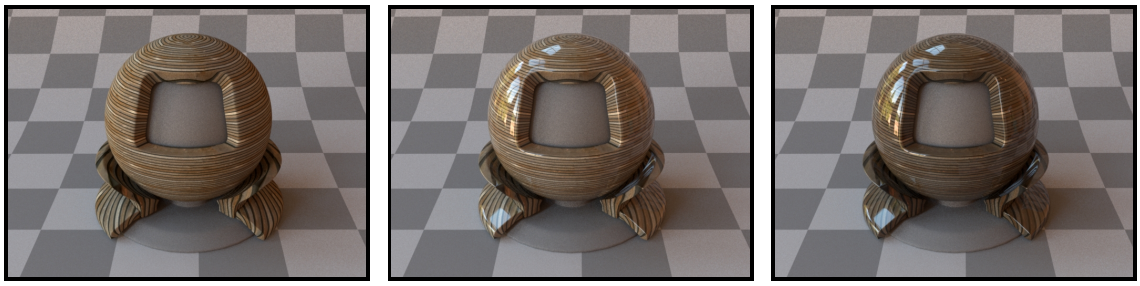
This plugin describes a smooth plastic-like material with internal scattering. It uses the Fresnel reflection and transmission coefficients to provide direction-dependent specular and diffuse components. Since it is simple, realistic, and fast, this model is often a better choice than the phong, ward, and roughplastic plugins when rendering smooth plastic-like materials.

For convenience, this model allows to specify IOR values either numerically, or based on a list of known materials (see Table 3 for an overview).

Note that this plugin is quite similar to what one would get by applying the coating plugin to the diffuse material. The main difference is that this plugin is significantly faster, while at the same time causing less variance. Furthermore, it accounts for multiple interreflections inside the material (read on for details), which avoids a serious energy loss problem of the aforementioned plugin combination.

```
<bsdf type="plastic">
    <srgb name="diffuseReflectance" value="#18455c"/>
    <float name="intIOR" value="1.9"/>
</bsdf>
```

Listing 20: A shiny material whose diffuse reflectance is specified using sRGB

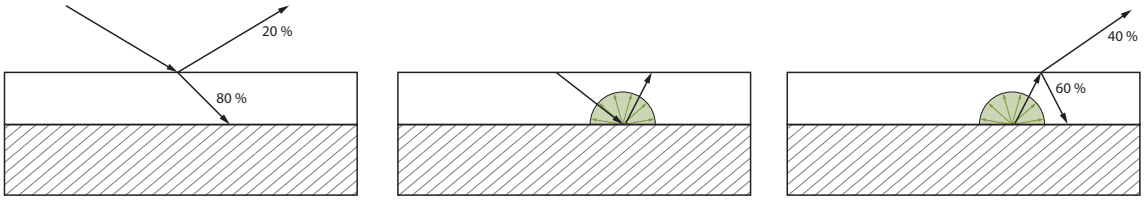


(a) Diffuse textured rendering (b) Plastic model, `nonlinear=false` (c) Plastic model, `nonlinear=true`

Figure 10: When asked to do so, this model can account for subtle nonlinear color shifts due to internal scattering processes. The above images show a textured object first rendered using `diffuse`, then `plastic` with the default parameters, and finally using `plastic` and support for nonlinear color shifts.

Internal scattering

Internally, this is model simulates the interaction of light with a diffuse base surface coated by a thin dielectric layer. This is a convenient abstraction rather than a restriction. In other words, there are many materials that can be rendered with this model, even if they might not not fit this description perfectly well.



(a) At the boundary, incident illumination is partly reflected and refracted (b) The refracted portion scatters diffusely at the base layer (c) Some of the illumination undergoes further internal scattering events

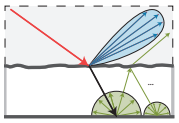
Figure 11: An illustration of the scattering events that are internally handled by this plugin

Given illumination that is incident upon such a material, a portion of the illumination is specularly reflected at the material boundary, which results in a sharp reflection in the mirror direction (Figure 11a). The remaining illumination refracts into the material, where it scatters from the diffuse base layer. (Figure 11b). While some of the diffusely scattered illumination is able to directly refract outwards again, the remainder is reflected from the interior side of the dielectric boundary and will in fact remain trapped inside the material for some number of internal scattering events until it is finally able to escape (Figure 11c).

Due to the mathematical simplicity of this setup, it is possible to work out the correct form of the model without actually having to simulate the potentially large number of internal scattering events.

Note that due to the internal scattering, the diffuse color of the material is in practice slightly different from the color of the base layer on its own—in particular, the material color will tend to shift towards darker colors with higher saturation. Since this can be counter-intuitive when using bitmap textures, these color shifts are disabled by default. Specify the parameter `nonlinear=true` to enable them. Figure 10 illustrates the resulting change. This effect is also seen in real life, for instance a piece of wood will look slightly darker after coating it with a layer of varnish.

8.2.9. Rough plastic material (`roughplastic`)



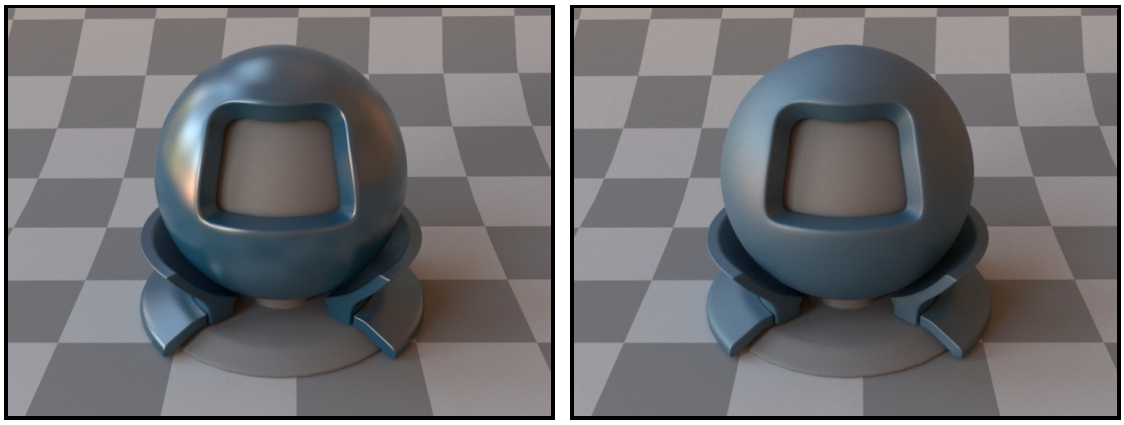
Parameter	Type	Description
`distribution`	`string`	Specifies the type of microfacet normal distribution used to model the surface roughness. (i) `beckmann`: Physically-based distribution derived from Gaussian random surfaces. This is the default. (ii) `ggx`: New distribution proposed by Walter et al. [48], which is meant to better handle the long tails observed in measurements of ground surfaces. Renderings with this distribution may converge slowly. (iii) `phong`: Classical $\cos^p\theta$ distribution. Due to the underlying microfacet theory, the use of this distribution here leads to more realistic behavior than the separately available phong plugin.
`alpha`	`float` or `texture`	Specifies the roughness of the unresolved surface microgeometry. When the Beckmann distribution is used, this parameter is equal to the *root mean square* (RMS) slope of the microfacets. (Default: 0.1).
`intIOR`	`float` or `string`	Interior index of refraction specified numerically or using a known material name. (Default: `polypropylene` / 1.49)
`extIOR`	`float` or `string`	Exterior index of refraction specified numerically or using a known material name. (Default: `air` / 1.000277)
`specular↙Reflectance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)
`diffuse↙Reflectance`	`spectrum` or `texture`	Optional factor used to modulate the diffuse reflection component (Default: 0.5)
`nonlinear`	`boolean`	Account for nonlinear color shifts due to internal scattering? See the `plastic` plugin for details. (Default: Don't account for them and preserve the texture colors, i.e. `false`)

This plugin implements a realistic microfacet scattering model for rendering rough dielectric materials with internal scattering, such as plastic. It can be interpreted as a fancy version of the Cook-Torrance model and should be preferred over heuristic models like `phong` and `ward` when possible.

Microfacet theory describes rough surfaces as an arrangement of unresolved and ideally specular facets, whose normal directions are given by a specially chosen *microfacet distribution*. By accounting for shadowing and masking effects between these facets, it is possible to reproduce the important off-specular reflections peaks observed in real-world measurements of such materials.

This plugin is essentially the "roughened" equivalent of the (smooth) plugin `plastic`. For very low values of α, the two will be identical, though scenes using this plugin will take longer to render due to the additional computational burden of tracking surface roughness.

For convenience, this model allows to specify IOR values either numerically, or based on a list of



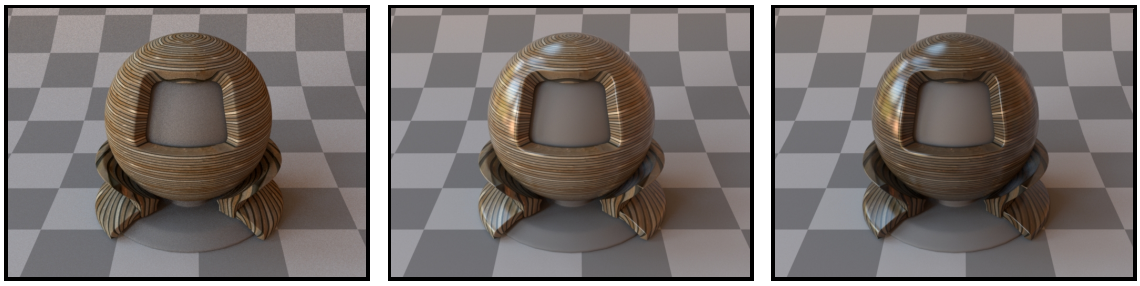
(a) Beckmann, $\alpha = 0.1$ (b) GGX, $\alpha = 0.3$

known materials (see Table 3 on page 58 for an overview). When no parameters are given, the plugin activates the defaults, which describe a white polypropylene plastic material with a light amount of roughness modeled using the Beckmann distribution.

Like the `plastic` material, this model internally simulates the interaction of light with a diffuse base surface coated by a thin dielectric layer (where the coating layer is now *rough*). This is a convenient abstraction rather than a restriction. In other words, there are many materials that can be rendered with this model, even if they might not not fit this description perfectly well.

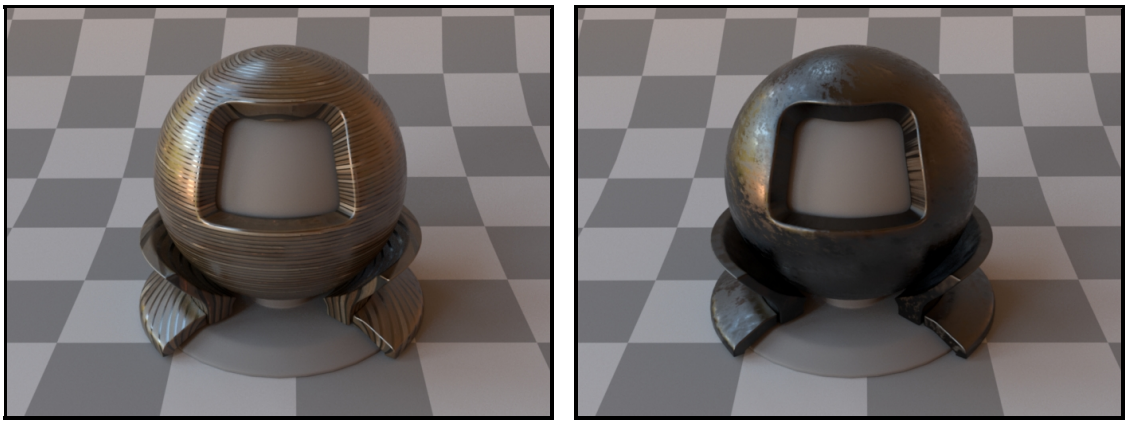
The simplicity of this setup makes it possible to account for interesting nonlinear effects due to internal scattering, which is controlled by the `nonlinear` parameter. For more details, please refer to the description of this parameter given in the the `plastic` plugin section on page 68.

To get an intuition about the effect of the surface roughness parameter α, consider the following approximate classification: a value of $\alpha = 0.001 - 0.01$ corresponds to a material with slight imperfections on an otherwise smooth surface finish, $\alpha = 0.1$ is relatively rough, and $\alpha = 0.3 - 0.7$ is *extremely* rough (e.g. an etched or ground finish). Values significantly above that are probably not too realistic.



(a) Diffuse textured rendering (b) Textured rough plastic model and `nonlinear=false` (c) Textured rough plastic model and `nonlinear=true`

Figure 12: When asked to do so, this model can account for subtle nonlinear color shifts due to internal scattering processes. The above images show a textured object first rendered using `diffuse`, then `roughplastic` with the default parameters, and finally using `roughplastic` and support for nonlinear color shifts.



(a) Wood material with smooth horizontal stripes

(b) A material with imperfections at a much smaller scale than what is modeled e.g. using a bump map.

Figure 13: The ability to texture the roughness parameter makes it possible to render materials with a structured finish, as well as "smudgy" objects.

```
<bsdf type="roughplastic">
    <string name="distribution" value="beckmann"/>
    <float name="intIOR" value="1.61"/>
    <spectrum name="diffuseReflectance" value="0"/>
    <!-- Fetch roughness values from a texture and slightly reduce them -->
    <texture type="scale" name="alpha">
        <texture name="alpha" type="bitmap">
            <string name="filename" value="roughness.png"/>
        </texture>
        <float name="scale" value="0.6"/>
    </texture>
</bsdf>
```

Listing 21: A material definition for black plastic material with a spatially varying roughness.

Technical details

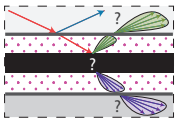
The implementation of this model is partly based on the paper "Microfacet Models for Refraction through Rough Surfaces" by Walter et al. [48]. Several different types of microfacet distributions are supported. Note that the choices are slightly more restricted here—in comparison to other rough scattering models in Mitsuba, anisotropic distributions are not allowed.

The implementation of this model makes heavy use of a *rough Fresnel transmittance* function, which is a generalization of the usual Fresnel transmittion coefficient to microfacet surfaces. Unfortunately, this function is normally prohibitively expensive, since each evaluation involves a numerical integration over the sphere.

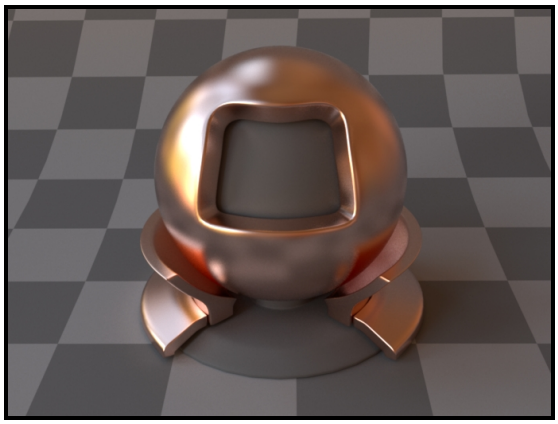
To avoid this performance issue, Mitsuba ships with data files (contained in the `data/microfacet` directory) containing precomputed values of this function over a large range of parameter values. At runtime, the relevant parts are extracted using tricubic interpolation.

When rendering with the Phong microfacet distributions, a conversion is used to turn the specified α roughness value into the Phong exponent. This is done in a way, such that the different distributions all produce a similar appearance for the same value of α.

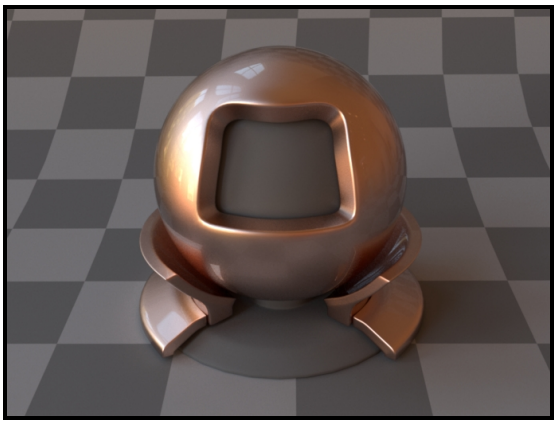
8.2.10. Smooth dielectric coating (`coating`)



Parameter	Type	Description
`intIOR`	`float` or `string`	Interior index of refraction specified numerically or using a known material name. (Default: `bk7` / 1.5046)
`extIOR`	`float` or `string`	Exterior index of refraction specified numerically or using a known material name. (Default: `air` / 1.000277)
`thickness`	`float`	Denotes the thickness of the layer (to model absorption — should be specified in inverse units of `sigmaA`) (Default: 1)
`sigmaA`	`spectrum` or `texture`	The absorption coefficient of the coating layer. (Default: 0, i.e. there is no absorption)
`specular↙ Reflectance`	`spectrum` or `texture`	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)
`(Nested plugin)`	`bsdf`	A nested BSDF model that should be coated.



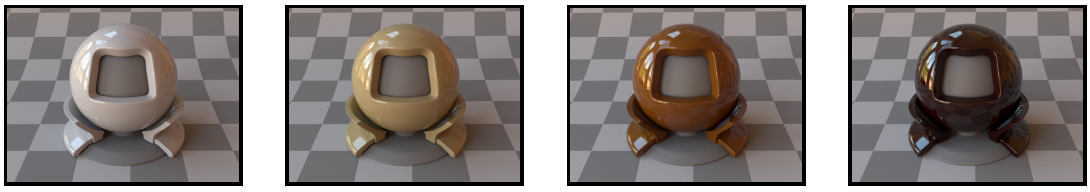
(a) Rough copper



(b) The same material coated with a single layer of clear varnish (see Listing 22)

This plugin implements a smooth dielectric coating (e.g. a layer of varnish) in the style of the paper "Arbitrarily Layered Micro-Facet Surfaces" by Weidlich and Wilkie [52]. Any BSDF in Mitsuba can be coated using this plugin, and multiple coating layers can even be applied in sequence. This allows designing interesting custom materials like car paint or glazed metal foil. The coating layer can optionally be tinted (i.e. filled with an absorbing medium), in which case this model also accounts for the directionally dependent absorption within the layer.

Note that the plugin discards illumination that undergoes internal reflection within the coating. This can lead to a noticeable energy loss for materials that reflect much of their energy near or below the critical angle (i.e. diffuse or very rough materials). Therefore, users are discouraged to use this plugin to coat smooth diffuse materials, since there is a separately available plugin named `plastic`, which covers the same case and does not suffer from energy loss.

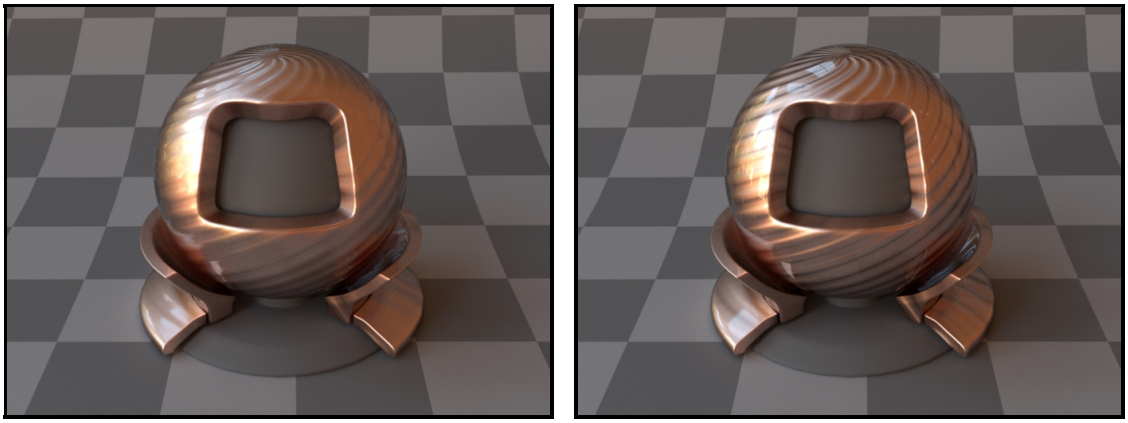


(a) thickness = 0 (b) thickness = 1 (c) thickness = 5 (d) thickness = 15

Figure 14: The effect of the layer thickness parameter on a tinted coating (sigmaT = (0.1, 0.2, 0.5))

```
<bsdf type="coating">
    <float name="intIOR" value="1.7"/>

    <bsdf type="roughconductor">
        <string name="material" value="Cu"/>
        <float name="alpha" value="0.1"/>
    </bsdf>
</bsdf>
```

Listing 22: Rough copper coated with a transparent layer of varnish



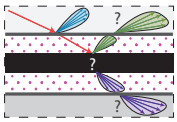
(a) Coated rough copper with a bump map applied on top (b) Bump mapped rough copper with a coating on top

Figure 15: Some interesting materials can be created simply by applying Mitsuba's material modifiers in different orders.

Technical details

Evaluating the internal component of this model entails refracting the incident and exitant rays through the dielectric interface, followed by querying the nested material with this modified direction pair. The result is attenuated by the two Fresnel transmittances and the absorption, if any.

8.2.11. Rough dielectric coating (roughcoating)

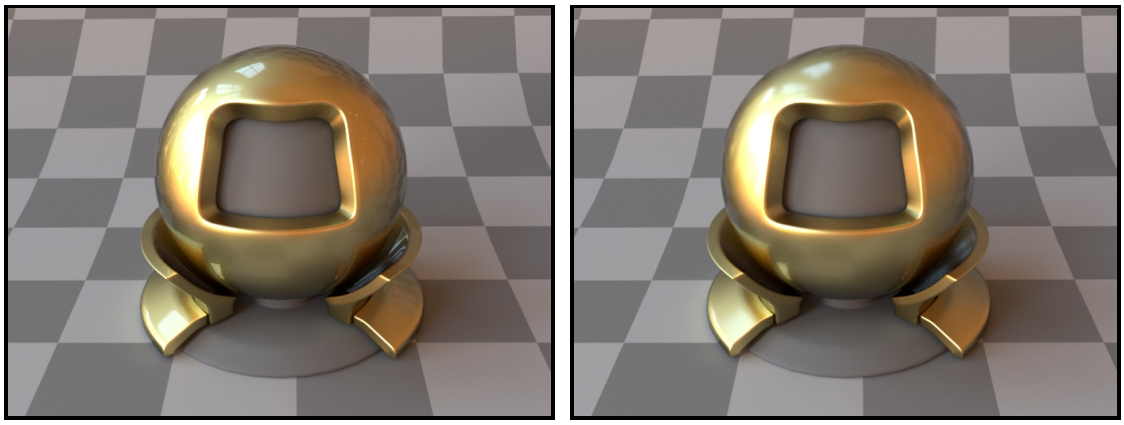


Parameter	Type	Description
distribution	string	Specifies the type of microfacet normal distribution used to model the surface roughness. (i) beckmann: Physically-based distribution derived from Gaussian random surfaces. This is the default. (ii) ggx: New distribution proposed by Walter et al. [48], which is meant to better handle the long tails observed in measurements of ground surfaces. Renderings with this distribution may converge slowly. (iii) phong: Classical $\cos^p\theta$ distribution. Due to the underlying microfacet theory, the use of this distribution here leads to more realistic behavior than the separately available phong plugin.
alpha	float or texture	Specifies the roughness of the unresolved surface microgeometry. When the Beckmann distribution is used, this parameter is equal to the *root mean square* (RMS) slope of the microfacets. (Default: 0.1).
intIOR	float or string	Interior index of refraction specified numerically or using a known material name. (Default: bk7 / 1.5046)
extIOR	float or string	Exterior index of refraction specified numerically or using a known material name. (Default: air / 1.000277)
thickness	float	Denotes the thickness of the layer (to model absorption — should be specified in inverse units of sigmaA) (Default: 1)
sigmaA	spectrum or texture	The absorption coefficient of the coating layer. (Default: 0, i.e. there is no absorption)
specular↙ Reflectance	spectrum or texture	Optional factor that can be used to modulate the specular reflection component. Note that for physical realism, this parameter should never be touched. (Default: 1.0)
(Nested plugin)	bsdf	A nested BSDF model that should be coated.

This plugin implements a *very* approximate[12] model that simulates a rough dielectric coating. It is essentially the roughened version of coating. Any BSDF in Mitsuba can be coated using this plugin and multiple coating layers can even be applied in sequence, which allows designing interesting custom materials. The coating layer can optionally be tinted (i.e. filled with an absorbing medium), in which case this model also accounts for the directionally dependent absorption within the layer.

Note that the plugin discards illumination that undergoes internal reflection within the coating. This can lead to a noticeable energy loss for materials that reflect much of their energy near or below

[12] The model only accounts for roughness in the specular reflection and Fresnel transmittance through the interface. The interior model receives incident illumination that is transformed *as if* the coating was smooth. While that's not quite correct, it is a convenient workaround when the coating plugin produces specular highlights that are too sharp.

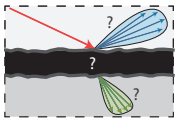


(a) Rough gold coated with a *smooth* varnish layer (b) Rough gold coated with a *rough* ($\alpha = 0.03$) varnish layer

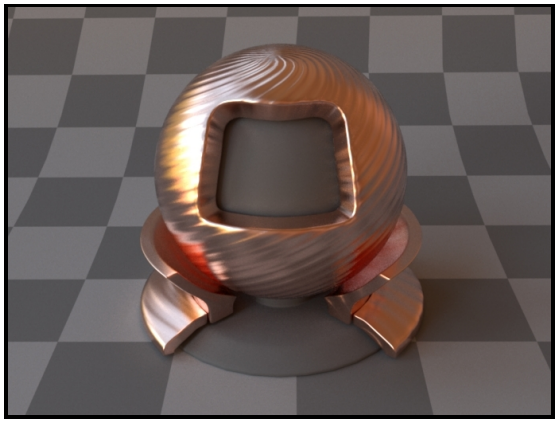
the critical angle (i.e. diffuse or very rough materials).

The implementation here is influenced by the paper "Arbitrarily Layered Micro-Facet Surfaces" by Weidlich and Wilkie [52].

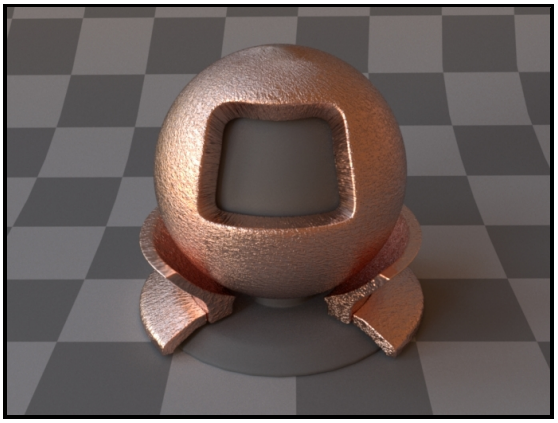
8.2.12. Bump map modifier (bumpmap)



Parameter	Type	Description
(Nested plugin)	texture	The luminance of this texture specifies the amount of displacement. The implementation ignores any constant offset—only changes in the luminance matter.
(Nested plugin)	bsdf	A BSDF model that should be affected by the bump map



(a) Bump map based on tileable diagonal lines



(b) An irregular bump map

Bump mapping [3] is a simple technique for cheaply adding surface detail to a rendering. This is done by perturbing the shading coordinate frame based on a displacement height field provided as a texture. This method can lend objects a highly realistic and detailed appearance (e.g. wrinkled or covered by scratches and other imperfections) without requiring any changes to the input geometry.

The implementation in Mitsuba uses the common approach of ignoring the usually negligible texture-space derivative of the base mesh surface normal. As side effect of this decision, it is invariant to constant offsets in the height field texture—only variations in its luminance cause changes to the shading frame.

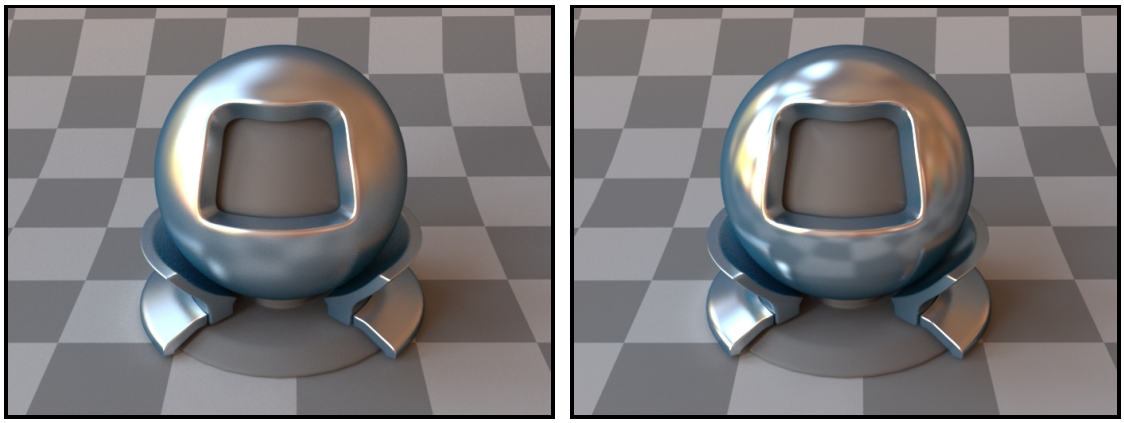
Note that the magnitude of the height field variations influences the strength of the displacement. If desired, the `scale` texture plugin can be used to magnify or reduce the effect of a bump map texture.

```
<bsdf type="bumpmap">
    <!-- The bump map is applied to a rough metal BRDF -->
    <bsdf type="roughconductor"/>

    <texture type="scale">
        <!-- The scale of the displacement gets multiplied by 10x -->
        <float name="scale" value="10"/>

        <texture type="bitmap">
            <string name="filename" value="bumpmap.png"/>
        </texture>
    </texture>
</bsdf>
```

Listing 23: A rough metal model with a scaled image-based bump map

8.2.13. Modified Phong BRDF (phong)

Parameter	Type	Description
exponent	float or texture	Specifies the Phong exponent (Default: 30).
specular↙ Reflectance	spectrum or texture	Specifies the weight of the specular reflectance component. (Default: 0.2)
diffuse↙ Reflectance	spectrum or texture	Specifies the weight of the diffuse reflectance component (Default: 0.5)



(a) Exponent = 60 (b) Exponent = 300

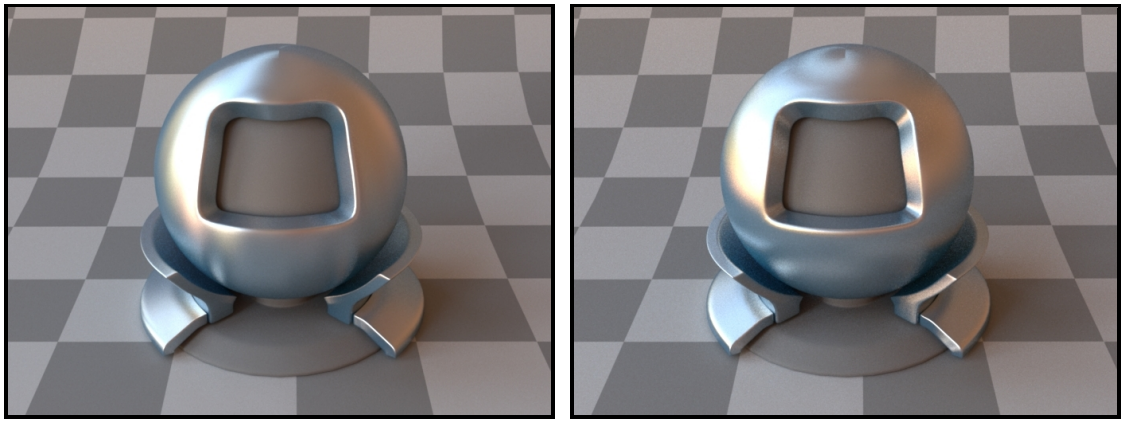
This plugin implements the modified Phong reflectance model as described in [37] and [30]. This heuristic model is mainly included for historical reasons—its use in new scenes is discouraged, since significantly more realistic models have been developed since 1975.

If possible, it is recommended to switch to a BRDF that is based on microfacet theory and includes knowledge about the material's index of refraction. In Mitsuba, two good alternatives to phong are the plugins roughconductor and roughplastic (depending on the material type).

When using this plugin, note that the diffuse and specular reflectance components should add up to a value less than or equal to one (for each color channel). Otherwise, they will automatically be scaled appropriately to ensure energy conservation.

8.2.14. Anisotropic Ward BRDF (`ward`)

Parameter	Type	Description
`variant`	`string`	Determines the variant of the Ward model to use: (i) `ward`: The original model by Ward [50] — suffers from energy loss at grazing angles. (ii) `ward-duer`: Corrected Ward model with lower energy loss at grazing angles [7]. Does not always conserve energy. (iii) `balanced`: Improved version of the `ward-duer` model with energy balance at all angles [11]. Default: `balanced`
`alphaU, alphaV`	`float` or `texture`	Specifies the anisotropic roughness values along the tangent and bitangent directions. (Default: 0.1).
`specular↙ Reflectance`	`spectrum` or `texture`	Specifies the weight of the specular reflectance component. (Default: 0.2)
`diffuse↙ Reflectance`	`spectrum` or `texture`	Specifies the weight of the diffuse reflectance component (Default: 0.5)



(a) $\alpha_u = 0.1,\ \alpha_v = 0.3$ (b) $\alpha_u = 0.3,\ \alpha_v = 0.1$

This plugin implements the anisotropic Ward reflectance model and several extensions. They are described in the papers

(i) "Measuring and Modeling Anisotropic Reflection" by Greg Ward [50]

(ii) "Notes on the Ward BRDF" by Bruce Walter [47]

(iii) "An Improved Normalization for the Ward Reflectance Model" by Arne Dür [7]

(iv) "A New Ward BRDF Model with Bounded Albedo" by Geisler-Moroder et al. [11]

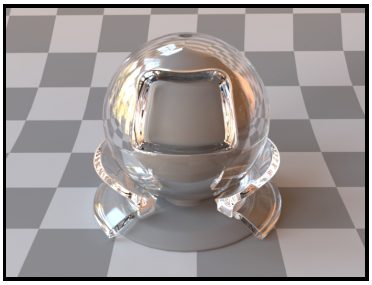
Like the Phong BRDF, the Ward model does not take the Fresnel reflectance of the material into account. In an experimental study by Ngan et al. [34], the Ward model performed noticeably worse than models based on microfacet theory.

For this reason, it is usually preferable to switch to a microfacet model that incorporates knowledge about the material's index of refraction. In Mitsuba, two such alternatives to `ward` are given by the plugins `roughconductor` and `roughplastic` (depending on the material type).

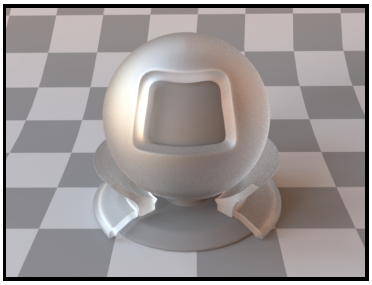
When using this plugin, note that the diffuse and specular reflectance components should add up to a value less than or equal to one (for each color channel). Otherwise, they will automatically be scaled appropriately to ensure energy conservation.

8.2.15. Mixture material (`mixturebsdf`)

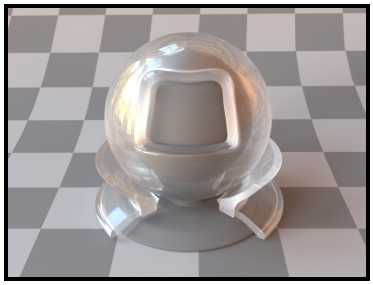
Parameter	Type	Description
`weights`	`string`	A comma-separated list of BSDF weights
(Nested plugin)	`bsdf`	Multiple BSDF instances that should be mixed according to the specified weights



(a) Smooth glass



(b) Rough glass



(c) An mixture of 70% smooth glass and 30% rough glass results in a more realistic smooth material with imperfections (Listing 24)

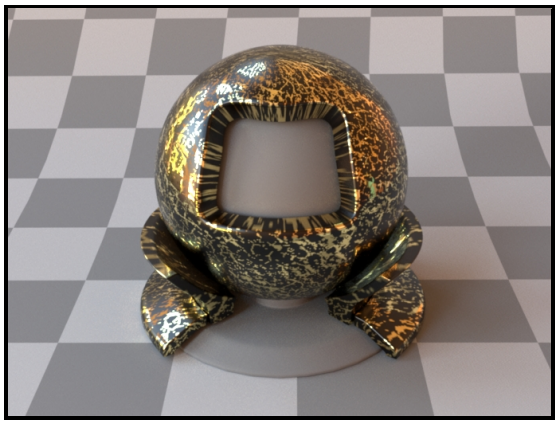
This plugin implements a "mixture" material, which represents linear combinations of multiple BSDF instances. Any surface scattering model in Mitsuba (be it smooth, rough, reflecting, or transmitting) can be mixed with others in this manner to synthesize new models. There is no limit on how many models can be mixed, but their combination weights must be non-negative and sum to a value of one or less to ensure energy balance. When they sum to less than one, the material will absorb a proportional amount of the incident illlumination.

```
<bsdf type="mixturebsdf">
    <string name="weights" value="0.7, 0.3"/>

    <bsdf type="dielectric"/>

    <bsdf type="roughdielectric">
        <float name="alpha" value="0.3"/>
    </bsdf>
</bsdf>
```

Listing 24: A material definition for a mixture of 70% smooth and 30% rough glass

8.2.16. Blended material (`blendbsdf`)

Parameter	Type	Description
`weight`	`float` or `texture`	A floating point value or texture with values between zero and one. The extreme values zero and one activate the first and second nested BSDF respectively, and inbetween values interpolate accordingly. (Default: 0.5)
(Nested plugin)	`bsdf`	Two nested BSDF instances that should be mixed according to the specified blending weight



(a) A material created by blending between dark rough plastic and smooth gold based on a binary bitmap texture (Listing 25)

This plugin implements a "blend" material, which represents linear combinations of two BSDF instances. It is conceptually very similar to the `mixturebsdf` plugin. The main difference is that `blendbsdf` can interpolate based on a texture rather than a set of constants.

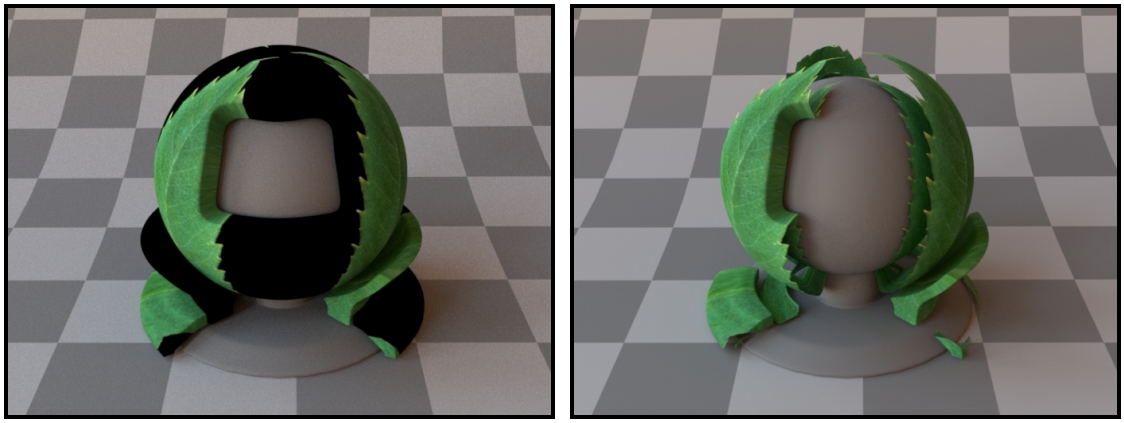
Any surface scattering model in Mitsuba (be it smooth, rough, reflecting, or transmitting) can be mixed with others in this manner to synthesize new models.

```
<bsdf type="blendbsdf">
    <texture name="weight" type="bitmap">
        <string name="wrapMode" value="repeat"/>
        <string name="filename" value="pattern.png"/>
    </texture>

    <bsdf type="conductor">
        <string name="material" value="Au"/>
    </bsdf>

    <bsdf type="roughplastic">
        <spectrum name="diffuseReflectance" value="0"/>
    </bsdf>
</bsdf>
```

Listing 25: Description of the material shown above

8.2.17. Opacity mask (`mask`)

Parameter	Type	Description
`opacity`	`spectrum` or `texture`	Specifies the per-channel opacity (where 1 = completely opaque) (Default: 0.5).
(Nested plugin)	`bsdf`	A base BSDF model that represents the non-transparent portion of the scattering



(a) Rendering without an opacity mask (b) Rendering *with* an opacity mask (Listing 26)

This plugin applies an opacity mask to add nested BSDF instance. It interpolates between perfectly transparent and completely opaque based on the `opacity` parameter.

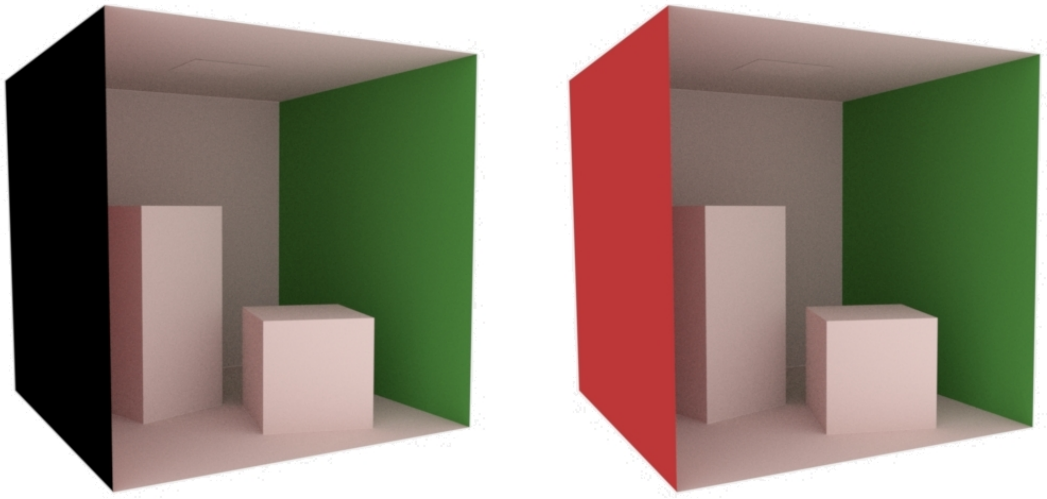
The transparency is internally implemented as a forward-facing Dirac delta distribution. Note that the standard `path` tracer does not have a good sampling strategy to deal with this, but the volumetric path tracer (`volpath`) does. It may thus be preferable when rendering scenes that contain the `mask` plugin, even if there is nothing "volumetric" in the scene.

```
<bsdf type="mask">
    <!-- Base material: a two-sided textured diffuse BSDF -->
    <bsdf type="twosided">
        <bsdf type="diffuse">
            <texture name="reflectance" type="bitmap">
                <string name="filename" value="leaf.png"/>
            </texture>
        </bsdf>
    </bsdf>

    <!-- Fetch the opacity mask from the alpha channel -->
    <texture name="opacity" type="bitmap">
        <string name="filename" value="leaf.png"/>
        <string name="channel" value="a"/>
    </texture>
</bsdf>
```

Listing 26: Material configuration for a transparent leaf

8.2.18. Two-sided BRDF adapter (`twosided`)

Parameter	Type	Description
(Nested plugin)	bsdf	A nested BRDF that should be turned into a two-sided scattering model. If two BRDFs are specified, they will be placed on the front and back side, respectively.



(a) From this angle, the Cornell box scene shows visible back-facing geometry

(b) Applying the `twosided` plugin fixes the rendering

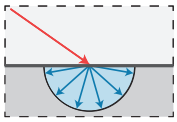
By default, all non-transmissive scattering models in Mitsuba are *one-sided* — in other words, they absorb all light that is received on the interior-facing side of any associated surfaces. Holes and visible back-facing parts are thus exposed as black regions.

Usually, this is a good idea, since it will reveal modeling issues early on. But sometimes one is forced to deal with improperly closed geometry, where the one-sided behavior is bothersome. In that case, this plugin can be used to turn one-sided scattering models into proper two-sided versions of themselves. The plugin has no parameters other than a required nested BSDF specification. It is also possible to supply two different BRDFs that should be placed on the front and back side, respectively.

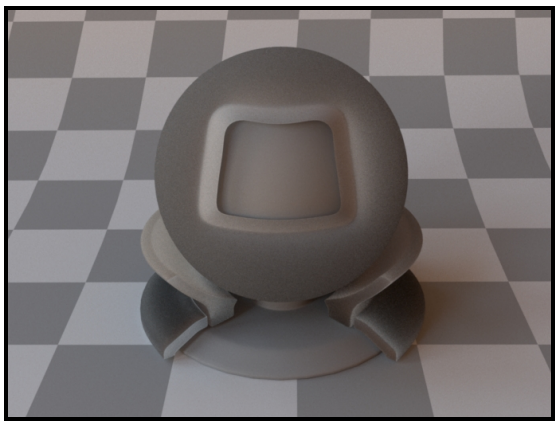
```
<bsdf type="twosided">
    <bsdf type="diffuse">
        <spectrum name="reflectance" value="0.4"/>
    </bsdf>
</bsdf>
```

Listing 27: A two-sided diffuse material

8.2.19. Diffuse transmitter (`difftrans`)



Parameter	Type	Description
`transmittance`	`spectrum` or `texture`	Specifies the diffuse transmittance of the material (Default: 0.5)

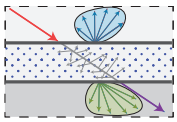


(a) The model with default parameters

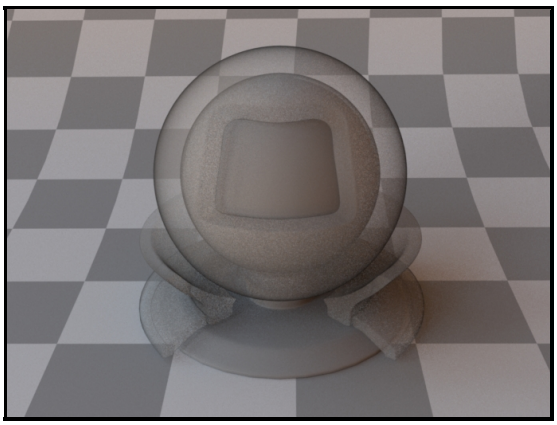
This BSDF models a non-reflective material, where any entering light loses its directionality and is diffusely scattered from the other side. This model can be combined[13] with a surface reflection model to describe translucent substances that have internal multiple scattering processes (e.g. plant leaves).

[13] For instance using the `mixturebsdf` plugin.

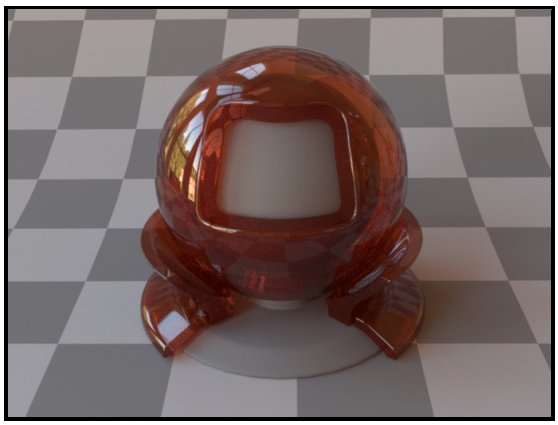
8.2.20. Hanrahan-Krueger BSDF (hk)



Parameter	Type	Description
material	string	Name of a material preset, see Table 5. (Default: skin1)
sigmaS	spectrum or texture	Specifies the scattering coefficient of the internal layer. (Default: based on material)
sigmaA	spectrum or texture	Specifies the absorption coefficient of the internal layer. (Default: based on material)
sigmaT & albedo	spectrum or texture	Optional: Alternatively, the scattering and absorption coefficients may also be specified using the extinction coefficient sigmaT and the single-scattering albedo. Note that only one of the parameter passing conventions can be used at a time (i.e. use either sigmaS&sigmaA *or* sigmaT&albedo)
thickness	float	Denotes the thickness of the layer. (should be specified in inverse units of sigmaA and sigmaS) (Default: 1)
(Nested plugin)	phase	A nested phase function instance that represents the type of scattering interactions occurring within the layer



(a) An index-matched scattering layer with parameters $\sigma_s = 2$, $\sigma_a = 0.1$, thickness= 0.1



(b) Example of the HK model with a dielectric coating (and the ketchup material preset, see Listing 28)

Figure 16: Renderings using the uncoated and coated form of the Hanrahan-Krueger model.

This plugin provides an implementation of the Hanrahan-Krueger BSDF [15] for simulating single scattering in thin index-matched layers filled with a random scattering medium. In addition, the implementation also accounts for attenuated light that passes through the medium without undergoing any scattering events.

This BSDF requires a phase function to model scattering interactions within the random medium. When no phase function is explicitly specified, it uses an isotropic one ($g = 0$) by default. A sample usage for instantiating the plugin is given on the next page:

```
<bsdf type="hk">
    <spectrum name="sigmaS" value="2"/>
    <spectrum name="sigmaA" value="0.1"/>
    <float name="thickness" value="0.1"/>

    <phase type="hg">
        <float name="g" value="0.8"/>
    </phase>
</bsdf>
```

When used in conjuction with the `coating` plugin, it is possible to model refraction and reflection at the layer boundaries when the indices of refraction are mismatched. The combination of these two plugins then reproduces the full model as it was originally proposed by Hanrahan and Krueger [15].

Note that this model does not account for light that undergoes multiple scattering events within the layer. This leads to energy loss, particularly at grazing angles, which can be seen in the left-hand image of Figure 16.

```
<bsdf type="coating">
    <float name="extIOR" value="1.0"/>
    <float name="intIOR" value="1.5"/>

    <bsdf type="hk">
        <string name="material" value="ketchup"/>
        <float name="thickness" value="0.01"/>
    </bsdf>
</bsdf>
```

Listing 28: A thin dielectric layer with measured ketchup scattering parameters

Note that when `sigmaS` = `sigmaA` = 0, or when `thickness=0`, any geometry associated with this BSDF becomes invisible, as light will pass through unchanged.

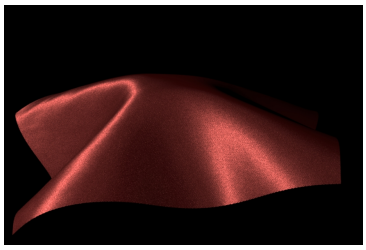
The implementation in Mitsuba is based on code by Tom Kazimiers and Marios Papas. Marios Papas has kindly verified the implementation of the coated and uncoated variants against both a path tracer and a separate reference implementation.

8.2.21. Irawan & Marschner woven cloth BRDF (`irawan`)

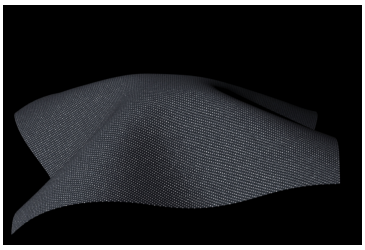
Parameter	Type	Description
`filename`	`string`	Path to a weave pattern description
`repeatU, repeatV`	`float`	Specifies the number of weave pattern repetitions over a $[0,1]^2$ region of the UV parameterization
(*Additional parameters*)	`spectrum` or `float`	Weave pattern files may define their own custom parameters; this is useful for instance to support changing the color of a weave without having to create a new file every time. These parameters must be specified directly to the plugin so that they can be appropriately resolved when the pattern file is loaded.

This plugin implements the Irawan & Marschner BRDF, a realistic model for rendering woven materials. This spatially-varying reflectance model uses an explicit description of the underlying weave pattern to create fine-scale texture and realistic reflections across a wide range of different weave types. To use the model, you must provide a special weave pattern file—for an example of what these look like, see the examples scenes available on the Mitsuba website.

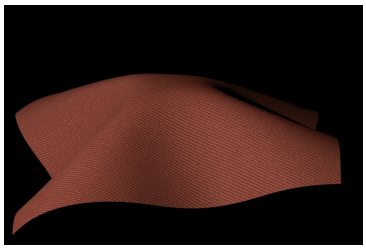
A detailed explanation of the model is beyond the scope of this manual. For reference, it is described in detail in the PhD thesis of Piti Irawan ("The Appearance of Woven Cloth" [17]). The code in Mitsuba a modified port of a previous Java implementation by Piti, which has been extended with a simple domain-specific weave pattern description language.



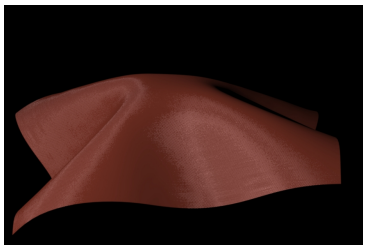
(a) Silk charmeuse



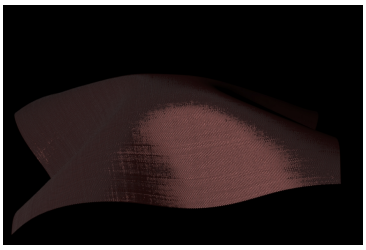
(b) Cotton denim



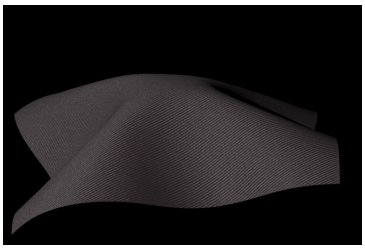
(c) Wool gabardine



(d) Polyester lining cloth



(e) Silk shantung



(f) Cotton twill

8.3. Textures

The following section describes the available texture data sources. In Mitsuba, textures are objects that can be attached to certain surface scattering model parameters to introduce spatial variation. In the documentation, these are listed as supporting the "`texture`" type. See Section 8.2 for many examples.

8.3.1. Bitmap texture (`bitmap`)

Parameter	Type	Description
`filename`	`string`	Filename of the bitmap to be loaded
`wrapMode`, `wrapModeU`, `wrapModeV`	`string`	Behavior of texture lookups outside of the $[0,1]$ *uv* range. (i) `repeat`: Repeat the texture indefinitely (ii) `mirror`: Mirror the texture along its boundaries (iii) `clamp`: Clamp *uv* coordinates to $[0,1]$ before a lookup (iv) `zero`: Switch to a zero-valued texture (v) `one`: Switch to a one-valued texture Default: `repeat`. The parameter `wrapMode` is a shortcut for setting both `wrapModeU` and `wrapModeV` at the same time.
`gamma`	`float`	Optional parameter to override the gamma value of the source bitmap, where 1 indicates a linear color space and the special value -1 corresponds to sRGB. (Default: automatically detect based on the image type and metadata)
`filterType`	`string`	Specifies the texture filtering that should be used for lookups (i) `ewa`: Elliptically weighted average (a.k.a. anisotropic filtering). This produces the best quality (ii) `trilinear`: Simple trilinear (isotropic) filtering. (iii) `nearest`: No filtering, do nearest neighbor lookups. Default: `ewa`.
`maxAnisotropy`	`float`	Specific to `ewa` filtering, this parameter limits the anisotropy (and thus the computational cost) of filtured texture lookups. The default of 20 is a good compromise.
`cache`	`boolean`	Preserve generated MIP map data in a cache file? This will cause a file named *filename*`.mip` to be created. (Default: automatic—use caching for textures larger than 1M pixels.)
`uoffset, voffset`	`float`	Numerical offset that should be applied to UV lookups
`uscale, vscale`	`float`	Multiplicative factors that should be applied to UV lookups
`channel`	`string`	Create a monochromatic texture based on one of the image channels (e.g. `r`, `g`, `b`, `a`, `x`, `y`, `z` etc.). (Default: use all channels)

This plugin provides a bitmap-backed texture source that supports *filtered* texture lookups on[14] JPEG, PNG, OpenEXR, RGBE, TGA, and BMP files. Filtered lookups are useful to avoid aliasing when rendering textures that contain high frequencies (see the next page for an example).

The plugin operates as follows: when loading a bitmap file, it is first converted into a linear color space. Following this, a MIP map is constructed that is necessary to perform filtered lookups during rendering. A *MIP map* is a hierarchy of progressively lower resolution versions of the input image,

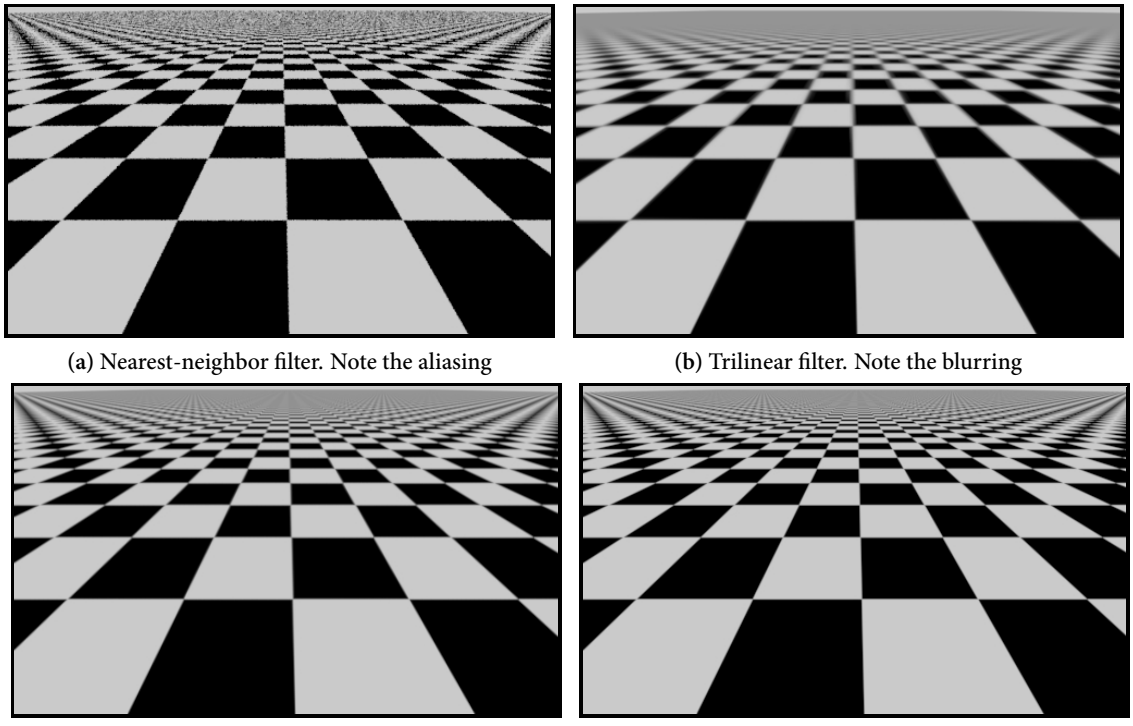
[14] Some of these may not be available depending on how Mitsuba was compiled.

where the resolution of adjacent levels differs by a factor of two. Mitsuba creates this hierarchy using Lanczos resampling to obtain very high quality results. Note that textures may have an arbitrary resolution and are not limited to powers of two. Three different filtering modes are supported:

(i) Nearest neighbor lookups effectively disable filtering and always query the highest-resolution version of the texture without any kind of interpolation. This is fast and requires little memory (no MIP map is created), but results in visible aliasing. Only a single pixel value is accessed.

(ii) The trilinear filter performs bilinear interpolation on two adjacent MIP levels and blends the results. Because it cannot do anisotropic (i.e. slanted) lookups in texture space, it must compromise either on the side of blurring or aliasing. The implementation in Mitsuba chooses blurring over aliasing (though note that (**b**) is an extreme case). Only 8 pixel values are accessed.

(iii) The EWA filter performs anisotropicically filtered lookups on two adjacent MIP map levels and blends them. This produces the best quality, but at the expense of computation time. Generally, 20-40 pixel values must be read for a single EWA texture lookup. To limit the number of pixel accesses, the `maxAnisotropy` parameter can be used to bound the amount of anisotropy that a texture lookup is allowed to have.



(**a**) Nearest-neighbor filter. Note the aliasing

(**b**) Trilinear filter. Note the blurring

(**c**) EWA filter

(**d**) Ground truth (512 samples per pixel)

Figure 17: A somewhat contrived comparison of the different filters when rendering a high-frequency checkerboard pattern using four samples per pixel. The EWA method (the default) pre-filters the texture anisotropically to limit blurring and aliasing, but has a higher computational cost than the other filters.

Caching and memory requirements: When a texture is read, Mitsuba internally converts it into an uncompressed linear format using a half precision (`float16`)-based representation. This is convenient for rendering but means that textures require copious amounts of memory (in particular, the size of the occupied memory region might be orders of magnitude greater than that of the original input file).

For instance, a basic 10 megapixel image requires as much as 76 MiB of memory! Loading, color space transformation, and MIP map construction require up to several seconds in this case. To reduce these overheads, Mitsuba 0.4.0 introduced MIP map caches. When a large texture is loaded for the first time, a MIP map cache file with the name *filename*`.mip` is generated. This is essentially a verbatim copy of the in-memory representation created during normal rendering. Storing this information as a separate file has two advantages:

(i) MIP maps do not have to be regenerated in subsequent Mitsuba runs, which substantially reduces scene loading times.

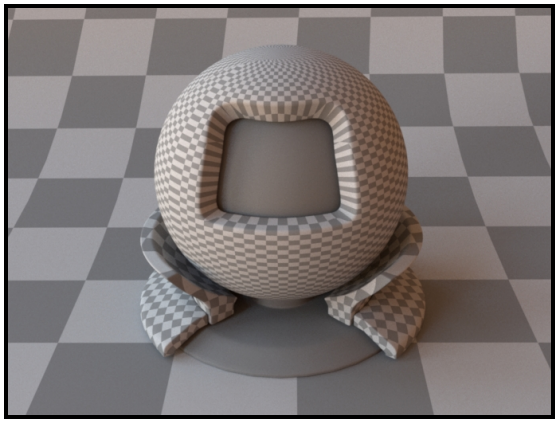
(ii) Because the texture storage is entirely disk-backed and can be *memory-mapped*, Mitsuba is able to work with truly massive textures that would otherwise exhaust the main system memory.

The texture caches are automatically regenerated when the input texture is modified. Of course, the cache files can be cumbersome when they are not needed anymore. On Linux or Mac OS, they can safely be deleted by executing the following command within a scene directory.

```
$ find . -name "*.mip" -delete
```

8.3.2. Checkerboard (`checkerboard`)

Parameter	Type	Description
`color0, color1`	`spectrum`	Color values for the two differently-colored patches (Default: 0.4 and 0.2)
`uoffset, voffset`	`float`	Numerical offset that should be applied to UV values before a lookup
`uscale, vscale`	`float`	Multiplicative factors that should be applied to UV values before a lookup

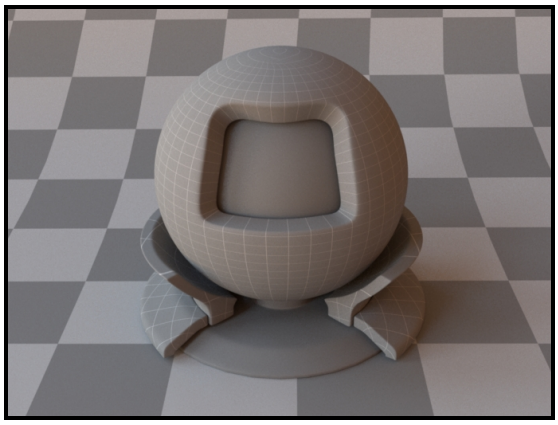


(a) Checkerboard applied to the material test object as well as the ground plane

This plugin implements a simple procedural checkerboard texture with customizable colors.

8.3.3. Procedural grid texture (`gridtexture`)

Parameter	Type	Description
`color0`	`spectrum`	Color values of the background (Default: 0.2)
`color1`	`spectrum`	Color value of the lines (Default: 0.4)
`lineWidth`	`float`	Width of the grid lines in UV space (Default: 0.01)
`uscale, vscale`	`float`	Multiplicative factors that should be applied to UV values before a lookup
`uoffset, voffset`	`float`	Numerical offset that should be applied to UV values before a lookup



(a) Grid texture applied to the material test object

This plugin implements a simple procedural grid texture with customizable colors and line width.

8.3.4. Scaling passthrough texture (`scale`)

Parameter	Type	Description
`value`	`spectrum` or `texture`	Specifies the spectrum or nested texture that should be scaled
`value`	`float`	Specifies the scale value

This simple plugin wraps a nested texture plugin and multiplies its contents by a user-specified value. This can be quite useful when a texture is too dark or too bright. The plugin can also be used to adjust the height of a bump map when using the `bumpmap` plugin.

```
<texture type="scale">
    <float name="scale" value="0.5"/>

    <texture type="bitmap">
        <string name="filename" value="wood.jpg"/>
    </texture>
</texture>
```

Listing 29: Scaling the contents of a bitmap texture

8.3.5. Vertex color passthrough texture (`vertexcolors`)

When rendering with a mesh that contains vertex colors, this plugin exposes the underlying color data as a texture. Currently, this is only supported by the PLY file format loader.

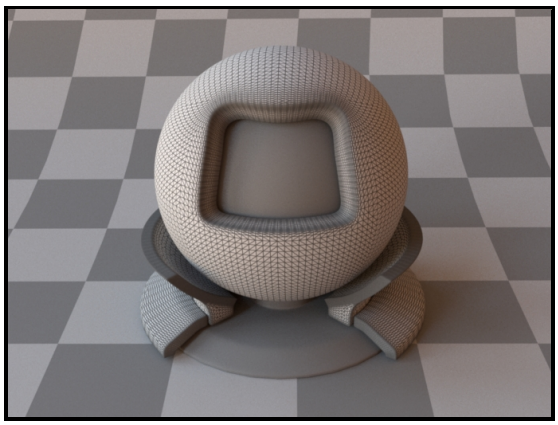
Here is an example:

```
<shape type="ply">
    <string name="filename" value="mesh.ply"/>

    <bsdf type="diffuse">
        <texture type="vertexcolors" name="reflectance"/>
    </bsdf>
</shape>
```

Listing 30: Rendering a PLY file with vertex colors

8.3.6. Wireframe texture (`wireframe`)

Parameter	Type	Description
`interiorColor`	`spectrum`	Color value of the interior of triangles (Default: 0.5)
`edgeColor`	`spectrum`	Edge color value (Default: 0.1)
`lineWidth`	`float`	World-space width of the mesh edges (Default: automatic)
`stepWidth`	`float`	Controls the width of of step function used for the color transition. It is specified as a value between zero and one (relative to the `lineWidth` parameter) (Default: 0.5)

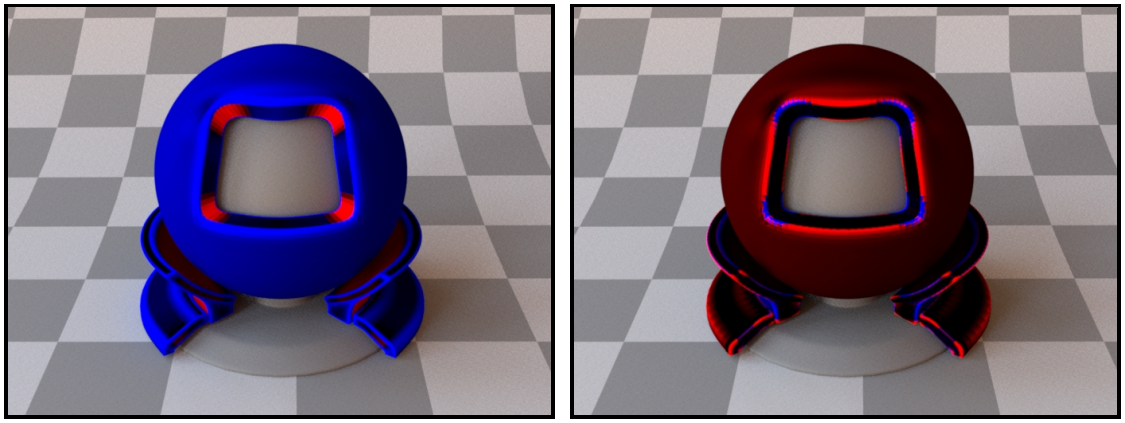


(a) Wireframe texture applied to the material test object

This plugin implements a simple two-color wireframe texture map that reveals the structure of a triangular mesh.

8.3.7. Curvature texture (`curvature`)

Parameter	Type	Description
`curvature`	`string`	Specifies what should be shown – must be equal to `mean` or `gaussian`.
`scale`	`float`	A scale factor to bring curvature values into the displayable range [-1, 1]. Everything outside of this range will be clamped.



(a) Mean curvature **(b)** Gaussian curvature

This texture can visualize the mean and Gaussian curvature of the underlying shape for inspection. Red and blue denote positive and negative values, respectively.

8.4. Subsurface scattering models

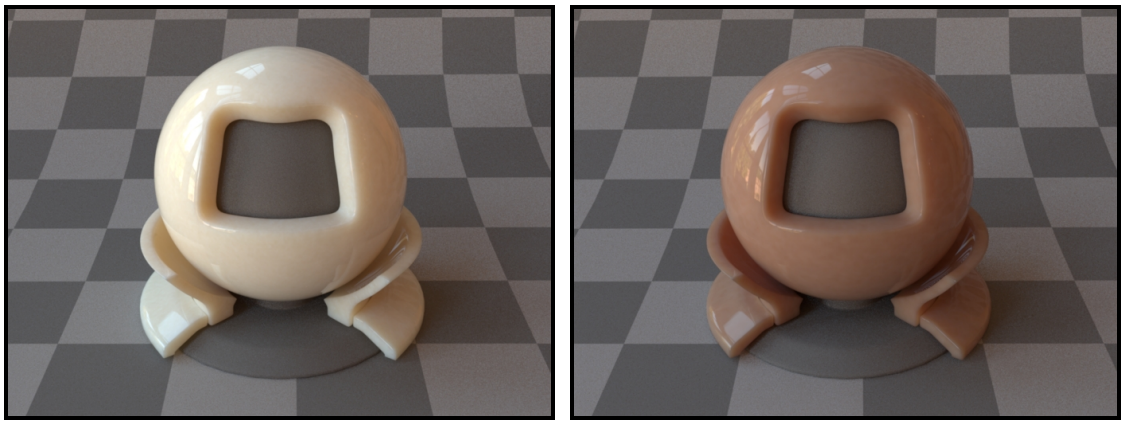
There are two ways of simulating subsurface scattering within Mitsuba: participating media and subsurface scattering models.

Subsurface scattering models: Described in this section. These can be thought of as a first-order approximation of what happens inside a participating medium. They are preferable when visually appealing output should be generated *quickly* and the demands on accuracy are secondary. At the moment, there is only one subsurface scattering model (the `dipole`), which is described on the next page.

Participating media: Described in Section 8.5. When modeling subsurface scattering using a participating medium, Mitsuba performs a *full* radiative transport simulation, which correctly accounts for all scattering events. This is more accurate but generally significantly slower.

8.4.1. Dipole-based subsurface scattering model (`dipole`)

Parameter	Type	Description
`material`	`string`	Name of a material preset, see Table 5. (Default: `skin1`)
`sigmaA, sigmaS`	`spectrum`	Absorption and scattering coefficients of the medium in inverse scene units. These parameters are mutually exclusive with `sigmaT` and `albedo` (Default: configured based on `material`)
`sigmaT, albedo`	`spectrum`	Extinction coefficient in inverse scene units and a (unitless) single-scattering albedo. These parameters are mutually exclusive with `sigmaA` and `sigmaS` (Default: configured based on `material`)
`scale`	`float`	Optional scale factor that will be applied to the `sigma*` parameters. It is provided for convenience when accomodating data based on different units, or to simply tweak the density of the medium. (Default: 1)
`intIOR`	`float` or `string`	Interior index of refraction specified numerically or using a known material name. (Default: based on `material`)
`extIOR`	`float` or `string`	Exterior index of refraction specified numerically or using a known material name. (Default: based on `material`)
`irrSamples`	`integer`	Number of samples to use when estimating the irradiance at a point on the surface (Default: 16)

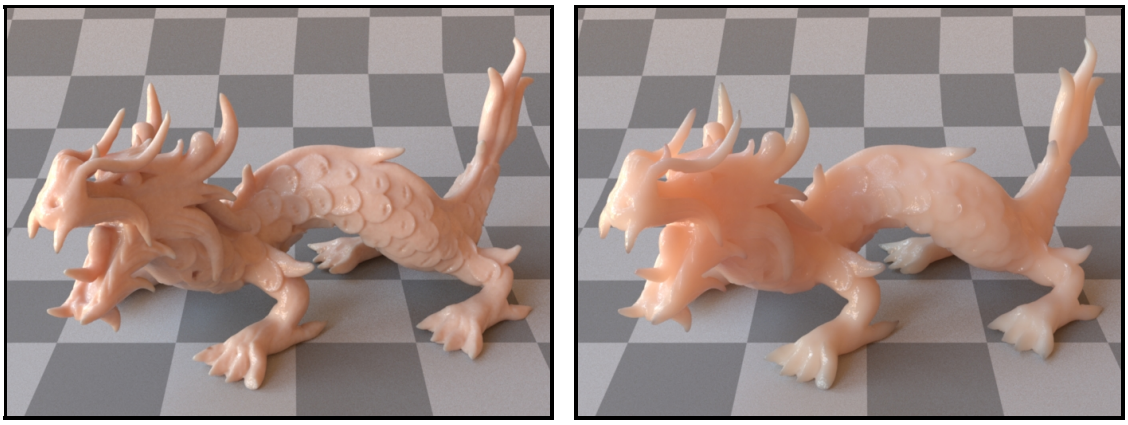


(a) The material test ball rendered with the `skimmilk` material preset

(b) The material test ball rendered with the `skin1` material preset

This plugin implements the classic dipole subsurface scattering model from radiative transport and medical physics [8, 9] in the form proposed by Jensen et al. [23]. It relies on the assumption that light entering a material will undergo many (i.e. hundreds) of internal scattering events, such that diffusion theory becomes applicable. In this case[15] a simple analytic solution of the subsurface scattering profile is available that enables simulating this effect without having to account for the vast

[15] and after making several fairly strong simplifications: the geometry is assumed to be a planar half-space, and the internal scattering from the material boundary is only considered approximately.



(a) `scale=1` (b) `scale=0.2`

Figure 18: The dragon model rendered with the `skin2` material preset (model courtesy of XYZ RGB). The `scale` parameter is useful to communicate the relative size of an object to the viewer.

numbers of internal scattering events individually.

For each `dipole` instance in the scene, the plugin adds a pre-process pass to the rendering that computes the irradiance on a large set of sample positions spread uniformly over the surface in question. The locations of these points are chosen using a technique by Bowers et al. [4] that creates particularly well-distributed (blue noise) samples. Later during rendering, these illumination samples are convolved with the diffusion profile using a fast hierarchical technique proposed by Jensen and Buhler [22].

There are two different ways of configuring the medium properties. One possibility is to load a material preset using the `material` parameter—see Table 5 for details. Alternatively, when specifying parameters by hand, they can either be provided using the scattering and absorption coefficients, or by declaring the extinction coefficient and single scattering albedo (whichever is more convenient). Mixing these parameter initialization methods is not allowed.

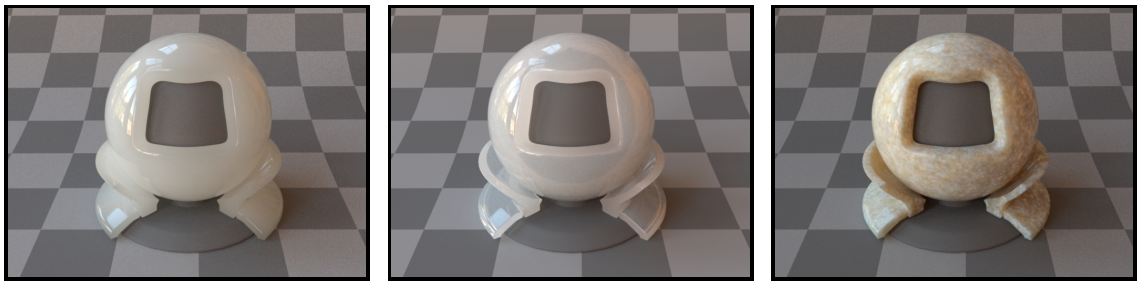
All scattering parameters (named `sigma*`) should be provided in inverse scene units. For instance, when a world-space distance of 1 unit corresponds to a meter, the scattering coefficents must be in units of inverse meters. For convenience, the `scale` parameter can be used to correct this. For instance, when the scene is in meters and the coefficients are in inverse millimeters, set `scale=1000`.

Note that a subsurface integrator can be associated with an `id` and shared by several shapes using the reference mechanism introduced in Section 6. This can be useful when an object is made up of many separate sub-shapes.

Typical material setup

To create a realistic material with subsurface scattering, it is necessary to associate the underlying shape with an appropriately configured surface and subsurface scattering model. Both should be aware of the material's index of refraction.

Because the `dipole` plugin is responsible for all internal scattering, the surface scattering model should only account for specular reflection due to the index of refraction change. There are two models in Mitsuba that can do this: `plastic` and `roughplastic` (for smooth and rough interfaces, respectively). An example is given on the next page.



(a) Rendered using `dipole` (b) Rendered using `homogeneous` (c) `irrSamples` set too low

Figure 19: Two problem cases that may occur when rendering with the `dipole`: (a)-(b): These two renderings show a glass ball filled with diluted milk rendered using diffusion theory and radiative transport, respectively. The former produces an incorrect result, since the assumption of many scattering events breaks down. (c): When the number of irradiance samples is too low when rendering with the dipole model, the resulting noise becomes visible as "blotchy" artifacts in the rendering.

```
<shape type="...">
    <subsurface type="dipole">
        <string name="intIOR" value="water"/>
        <string name="extIOR" value="air"/>
        <rgb name="sigmaS" value="87.2, 127.2, 143.2"/>
        <rgb name="sigmaA" value="1.04, 5.6, 11.6"/>
        <integer name="irrSamples" value="64"/>
    </subsurface>

    <bsdf type="plastic">
        <string name="intIOR" value="water"/>
        <string name="extIOR" value="air"/>
        <!-- Note: the diffuse component must be disabled! -->
        <spectrum name="diffuseReflectance" value="0"/>
    </bsdf>
<shape>
```

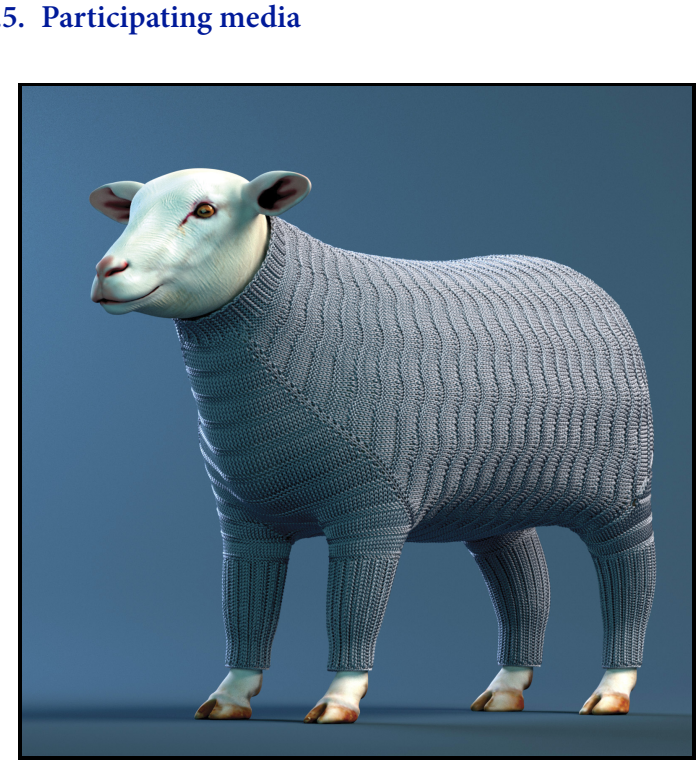
Remarks:

- This plugin only implements the multiple scattering component of the dipole model, i.e. single scattering is omitted. Furthermore, the numerous assumptions built into the underlying theory can cause severe inaccuracies.

  For this reason, this plugin is the right choice for making pictures that "look nice", but it should be avoided when the output must hold up to real-world measurements. In this case, please use participating media (Section 8.5).

- It is quite important that the `sigma*` parameters have the right units. For instance: if the `sigmaT` parameter is accidentally set to a value that is too small by a factor of 1000, the plugin will attempt to create one million times as many irradiance samples, which will likely cause the rendering process to crash with an "out of memory" failure.

8.5. Participating media



(a) A knitted sheep sweater (Ridged Feather pattern)

(b) A knitted sweater for an alien character (Braid Cables pattern)

Figure 20: Participating media are not limited to smoke or fog: they are also great for rendering fuzzy materials such as these knitted sweaters (made using the `heterogeneous` and `microflake` plugins). Figure courtesy of Yuksel et al. [53], models courtesy of Rune Spaans and Christer Sveen.

In Mitsuba, participating media are used to simulate materials ranging from fog, smoke, and clouds, over translucent materials such as skin or milk, to "fuzzy" structured substances such as woven or knitted cloth.

This section describes the two available types of media (`homogeneous` and `heterogeneous`). In pratice, these will be combined with a phase function, which are described in Section 8.6. Participating media are usually also attached to shapes in the scene. How this is done is described at the beginning of Section 8.1 on page 33.

When a medium permeates a volume of space (e.g. fog) that includes sensors or emitters, it is important to assign the medium to them. This can be done using the referencing mechanism:

```
<medium type="homogeneous" id="fog">
    <!-- .... homogeneous medium parameters .... -->
</medium>

<sensor type="perspective">
    <!-- .... perspective camera parameters .... -->

    <!-- Reference the fog medium from within the sensor declaration
         to make it aware that it is embedded inside this medium -->
    <ref id="fog"/>
</sensor>
```

8.5.1. Homogeneous participating medium (homogeneous)

Parameter	Type	Description
material	string	Name of a material preset, see Table 5. (Default: skin1)
sigmaA, sigmaS	spectrum	Absorption and scattering coefficients of the medium in inverse scene units. These parameters are mutually exclusive with sigmaT and albedo (Default: configured based on material)
sigmaT, albedo	spectrum	Extinction coefficient in inverse scene units and a (unitless) single-scattering albedo. These parameters are mutually exclusive with sigmaA and sigmaS (Default: configured based on material)
scale	float	Optional scale factor that will be applied to the sigma* parameters. It is provided for convenience when accomodating data based on different units, or to simply tweak the density of the medium. (Default: 1)
(Nested plugin)	phase	A nested phase function that describes the directional scattering properties of the medium. When none is specified, the renderer will automatically use an instance of isotropic.

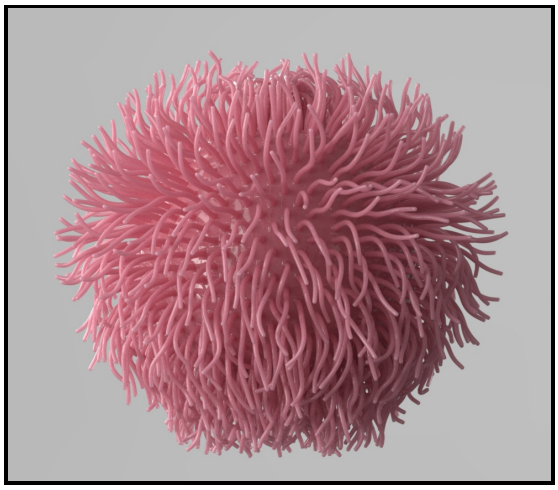
This class implements a flexible homogeneous participating medium with support for arbitrary phase functions and various medium sampling methods. It provides two different ways of configuring the medium properties. One possibility is to load a material preset using the material parameter—see Table 5 for details. Alternatively, when specifying parameters by hand, they can either be provided using the scattering and absorption coefficients, or by declaring the extinction coefficient and single scattering albedo (whichever is more convenient). Mixing these parameter initialization methods is not allowed.

All scattering parameters (named sigma*) should be provided in inverse scene units. For instance, when a world-space distance of 1 unit corresponds to a meter, the scattering coefficents should have units of inverse meters. For convenience, the scale parameter can be used to correct the units. For instance, when the scene is in meters and the coefficients are in inverse millimeters, set scale to 1000.

```
<medium id="myMedium" type="homogeneous">
    <spectrum name="sigmaS" value="1"/>
    <spectrum name="sigmaA" value="0.05"/>

    <phase type="hg">
        <float name="g" value="0.7"/>
    </phase>
</medium>
```

Listing 31: Declaration of a forward scattering medium with high albedo

Note: Rendering media that have a spectrally varying extinction coefficient can be tricky, since all commonly used medium sampling methods suffer from high variance in that case. Here, it may often make more sense to render several monochromatic images separately (using only the coefficients for



(a) A squishy ball rendered with subsurface scattering and a dielectric BSDF (courtesy of Chanxi Zheng)

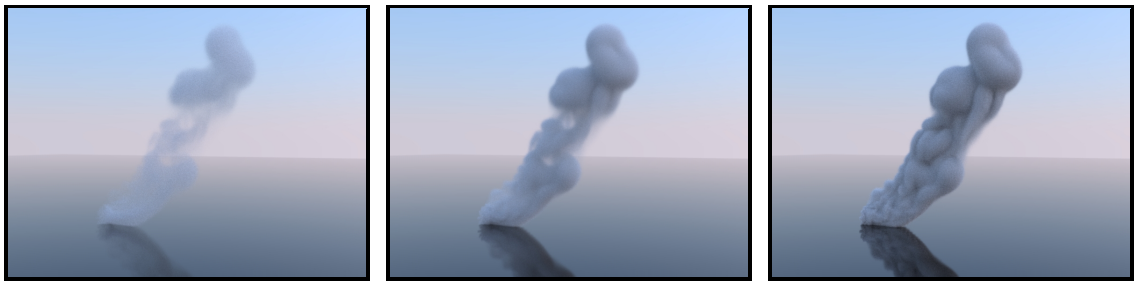
a single channel) and then merge them back into a RGB image. There is a `mtsutil` (Section 5.3.2) plugin named `joinrgb` that will perform this RGB merging process.

Name	Name	Name
Apple	Chicken1	Chicken2
Cream	Ketchup	Potato
Skimmilk	Skin1	Skin2
Spectralon	Wholemilk	
Lowfat Milk	Gatorade	White Grapefruit Juice
Reduced Milk	Chardonnay	Shampoo
Regular Milk	White Zinfandel	Strawberry Shampoo
Espresso	Merlot	Head & Shoulders Shampoo
Mint Mocha Coffee	Budweiser Beer	Lemon Tea Powder
Lowfat Soy Milk	Coors Light Beer	Orange Juice Powder
Regular Soy Milk	Clorox	Pink Lemonade Powder
Lowfat Chocolate Milk	Apple Juice	Cappuccino Powder
Regular Chocolate Milk	Cranberry Juice	Salt Powder
Coke	Grape Juice	Sugar Powder
Pepsi	Ruby Grapefruit Juice	Suisse Mocha
Sprite		

Table 5: This table lists all supported medium material presets. The top entries are from Jensen et al. [23], and the bottom ones are from Narasimhan et al. [33]. They all use units of $\frac{1}{mm}$, so remember to set `scale` appropriately when your scene is not in units of millimeters. These material presets can be used with the plugins `homogeneous`, `dipole`, and `hk`

8.5.2. Heterogeneous participating medium (`heterogeneous`)

Parameter	Type	Description
`method`	`string`	Specifies the sampling method that is used to generate scattering events within the medium. (i) `simpson`: Sampling is done by inverting a deterministic quadrature rule based on composite Simpson integration over small ray segments. Benefits from the use of good sample generators (e.g. `ldsampler`). (ii) `woodcock`: Generate samples using Woodcock tracking. This is usually faster and always unbiased, but has the disadvantages of not benefiting from good sample generators and not providing information that is required by bidirectional rendering techniques. Default: `woodcock`
`density`	`volume`	Volumetric data source that supplies the medium densities (in inverse scene units)
`albedo`	`volume`	Volumetric data source that supplies the single-scattering albedo
`orientation`	`volume`	Optional: volumetric data source that supplies the local particle orientations throughout the medium
`scale`	`float`	Optional scale factor that will be applied to the `density` parameter. Provided for convenience when accomodating data based on different units, or to simply tweak the density of the medium. (Default: 1)
`(Nested plugin)`	`phase`	A nested phase function that describes the directional scattering properties of the medium. When none is specified, the renderer will automatically use an instance of `isotropic`.



(a) 40 (b) 200 (c) 1000

Figure 21: Renderings of an index-matched medium using different scale factors (Listing 32)

This plugin provides a flexible heterogeneous medium implementation, which acquires its data from nested `volume` instances. These can be constant, use a procedural function, or fetch data from disk, e.g. using a memory-mapped density grid. See Section 8.7 for details on volume data sources.

Instead of allowing separate volumes to be provided for the scattering and absorption parameters `sigmaS` and `sigmaA` (as is done in homogeneous), this class instead takes the approach of enforcing a spectrally uniform value of `sigmaT`, which must be provided using a nested scalar-valued volume named `density`.

Another nested spectrum-valued `albedo` volume must also be provided, which is used to compute the scattering coefficient σ_s using the expression σ_s = `scale*density*albedo` (i.e. 'albedo' contains the single-scattering albedo of the medium.

Optionally, one can also provide an vector-valued `orientation` volume, which contains local particle orientation that will be passed to scattering models that support this, such as a the Microflake or Kajiya-Kay phase functions.

```
<!-- Declare a heterogeneous participating medium named 'smoke' -->
<medium type="heterogeneous" id="smoke">
    <string name="method" value="simpson"/>

    <!-- Acquire density values from an external data file -->
    <volume name="density" type="gridvolume">
        <string name="filename" value="frame_0150.vol"/>
    </volume>

    <!-- The albedo is constant and set to 0.9 -->
    <volume name="albedo" type="constvolume">
        <spectrum name="value" value="0.9"/>
    </volume>

    <!-- Use an isotropic phase function -->
    <phase type="isotropic"/>

    <!-- Scale the density values as desired -->
    <float name="scale" value="200"/>
 </medium>

<!-- Attach the index-matched medium to a shape in the scene -->
<shape type="obj">
    <!-- Load an OBJ file, which contains a mesh version
         of the axis-aligned box of the volume data file -->
    <string name="filename" value="bounds.obj"/>

    <!-- Reference the medium by ID -->
    <ref name="interior" id="smoke"/>

    <!-- If desired, this shape could also declare
         a BSDF to create an index-mismatched
         transition, e.g.

    <bsdf type="dielectric"/>
    -->
</shape>
```

Listing 32: A simple heterogeneous medium backed by a grid volume

8.6. Phase functions

This section contains a description of all implemented medium scattering models, which are also known as *phase functions*. These are very similar in principle to surface scattering models (or *BSDF*s), and essentially describe where light travels after hitting a particle within the medium.

The most commonly used models for smoke, fog, and other homogeneous media are isotropic scattering (`isotropic`) and the Henyey-Greenstein phase function (`hg`). Mitsuba also supports *anisotropic* media, where the behavior of the medium changes depending on the direction of light propagation (e.g. in volumetric representations of fabric). These are the Kajiya-Kay (`kkay`) and Microflake (`microflake`) models.

Finally, there is also a phase function for simulating scattering in planetary atmospheres (`rayleigh`).

8.6.1. Isotropic phase function (`isotropic`)

(a) Isotropic (b) Anisotropic micro-flakes

Figure 22: Heterogeneous volume renderings of a scarf model with isotropic and anisotropic phase functions.

This phase function simulates completely uniform scattering, where all directionality is lost after a single scattering interaction. It does not have any parameters.

8.6.2. Henyey-Greenstein phase function (`hg`)

Parameter	Type	Description
`g`	`float`	This parameter must be somewhere in the range −1 to 1 (but not equal to −1 or 1). It denotes the *mean cosine* of scattering interactions. A value greater than zero indicates that medium interactions predominantly scatter incident light into a similar direction (i.e. the medium is *forward-scattering*), whereas values smaller than zero cause the medium to be scatter more light in the opposite direction.

This plugin implements the phase function model proposed by Henyey and Greenstein [16]. It is parameterizable from backward- ($g < 0$) through isotropic- ($g = 0$) to forward ($g > 0$) scattering.

8.6.3. Rayleigh phase function (`rayleigh`)

Scattering by particles that are much smaller than the wavelength of light (e.g. individual molecules in the atmosphere) is well-approximated by the Rayleigh phase function. This plugin implements an unpolarized version of this scattering model (i.e the effects of polarization are ignored). This plugin is useful for simulating scattering in planetary atmospheres.

This model has no parameters.

8.6.4. Kajiya-Kay phase function (`kkay`)

This plugin implements the Kajiya-Kay [25] phase function for volumetric rendering of fibers, e.g. hair or cloth.

The function is normalized so that it has no energy loss when `ks`=1 and illumination arrives perpendicularly to the surface.

8.6.5. Micro-flake phase function (`microflake`)

Parameter	Type	Description
`stddev`	`float`	Standard deviation of the micro-flake normals. This specifies the roughness of the fibers in the medium.

(a) `stddev=0.2` (b) `stddev=0.05`

This plugin implements the anisotropic micro-flake phase function described in "A radiative transfer framework for rendering materials with anisotropic structure" by Wenzel Jakob, Adam Arbree, Jonathan T. Moon, Kavita Bala, and Steve Marschner [18].

The implementation in this plugin is specific to rough fibers and uses a Gaussian-type flake distribution. It is much faster than the spherical harmonics approach proposed in the original paper. This distribution, as well as the implemented sampling method, are described in the paper "Building Volumetric Appearance Models of Fabric using Micro CT Imaging" by Shuang Zhao, Wenzel Jakob, Steve Marschner, and Kavita Bala [54].

Note: this phase function must be used with a medium that specifies the local fiber orientation at different points in space. Please refer to `heterogeneous` for details.

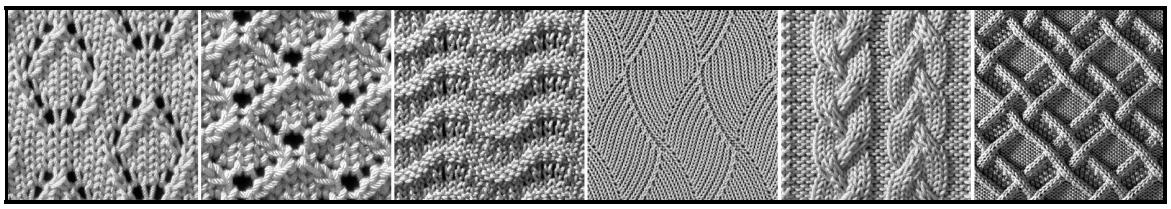


Figure 23: A range of different knit patterns, rendered using the `heterogeneous` and `microflake` plugins. Courtesy of Yuksel et al. [53].

8.6.6. Mixture phase function (`mixturephase`)

Parameter	Type	Description
`weights`	`string`	A comma-separated list of phase function weights
(Nested plugin)	`phase`	Multiple phase function instances that should be mixed according to the specified weights

This plugin implements a "mixture" scattering model analogous to `mixturebsdf`, which represents linear combinations of multiple phase functions. There is no limit on how many phase function can be mixed, but their combination weights must be non-negative and sum to a value of one or less to ensure energy balance.

8.7. Volume data sources

This section covers the different types of volume data sources included with Mitsuba. These plugins are intended to be used together with the `heterogeneous` medium plugin and provide three-dimensional spatially varying density, albedo, and orientation fields.

8.7.1. Constant-valued volume data source (`constvolume`)

Parameter	Type	Description
`value`	`float` or `spectrum` or `vector`	Specifies the value of the volume

This plugin provides a volume data source that is constant throughout its domain. Depending on how it is used, its value can either be a scalar, a color spectrum, or a 3D vector.

```
<medium type="heterogeneous">
    <volume type="constvolume" name="density">
        <float name="value" value="1"/>
    </volume>
    <volume type="constvolume" name="albedo">
        <rgb name="value" value="0.9 0.9 0.7"/>
    </volume>
    <volume type="constvolume" name="orientation">
        <vector name="value" x="0" y="1" z="0"/>
    </volume>

    <!-- .... remaining parameters for
         the 'heterogeneous' plugin .... -->
</medium>
```

Listing 33: Definition of a heterogeneous medium with homogeneous contents

8.7.2. Grid-based volume data source (`gridvolume`)

Parameter	Type	Description
`filename`	`string`	Specifies the filename of the volume data file to be loaded
`sendData`	`boolean`	When this parameter is set to `true`, the implementation will send all volume data to other network render nodes. Otherwise, they are expected to have access to an identical volume data file that can be mapped into memory. (Default: `false`)
`toWorld`	`transform`	Optional linear transformation that should be applied to the data
`min, max`	`point`	Optional parameter that can be used to re-scale the data so that it lies in the bounding box between `min` and `max`.

This class implements access to memory-mapped volume data stored on a 3D grid using a simple binary exchange format. The format uses a little endian encoding and is specified as follows:

Position	Content
Bytes 1-3	ASCII Bytes 'V', '0', and 'L'
Byte 4	File format version number (currently 3)
Bytes 5-8	Encoding identifier (32-bit integer). The following choices are available: 1. Dense `float32`-based representation 2. Dense `float16`-based representation (*currently not supported by this implementation*) 3. Dense `uint8`-based representation (The range 0..255 will be mapped to 0..1) 4. Dense quantized directions. The directions are stored in spherical coordinates with a total storage cost of 16 bit per entry.
Bytes 9-12	Number of cells along the X axis (32 bit integer)
Bytes 13-16	Number of cells along the Y axis (32 bit integer)
Bytes 17-20	Number of cells along the Z axis (32 bit integer)
Bytes 21-24	Number of channels (32 bit integer, supported values: 1 or 3)
Bytes 25-48	Axis-aligned bounding box of the data stored in single precision (order: xmin, ymin, zmin, xmax, ymax, zmax)
Bytes 49-*	Binary data of the volume stored in the specified encoding. The data are ordered so that the following C-style indexing operation makes sense after the file has been mapped into memory: `data[((zpos*yres + ypos)*xres + xpos)*channels + chan]` where `(xpos, ypos, zpos, chan)` denotes the lookup location.

Note that Mitsuba expects that entries in direction volumes are either zero or valid unit vectors.

When using this data source to represent floating point density volumes, please ensure that the values are all normalized to lie in the range $[0,1]$—otherwise, the Woodcock-Tracking integration method in `heterogeneous` will produce incorrect results.

8.7.3. Caching volume data source (`volcache`)

Parameter	Type	Description
`blockSize`	`integer`	Size of the individual cache blocks (Default: 8, i.e. $8 \times 8 \times 8$)
`voxelWidth`	`float`	Width of a voxel (in a cache block) expressed in world-space units. (Default: set to the ray marching step size of the nested medium)
`memoryLimit`	`integer`	Maximum allowed memory usage in MiB. (Default: 1024, i.e. 1 GiB)
`toWorld`	`transform`	Optional linear transformation that should be applied to the volume data
`(Nested plugin)`	`volume`	A nested volume data source

This plugin can be added between the renderer and another data source, for which it caches all data lookups using a LRU scheme. This is useful when the nested volume data source is expensive to evaluate.

The cache works by performing on-demand rasterization of subregions of the nested volume into blocks ($8 \times 8 \times 8$ by default). These are kept in memory until a user-specifiable threshold is exeeded, after which point a *least recently used* (LRU) policy removes records that haven't been accessed in a long time.

8.8. Emitters

Mitsuba supports a wide range of emitters/light sources, which can be classified into two main categories: emitters which are located somewhere within the scene, and emitters that *surround* the scene to simulate a distant environment. An overview of the available types is shown below:

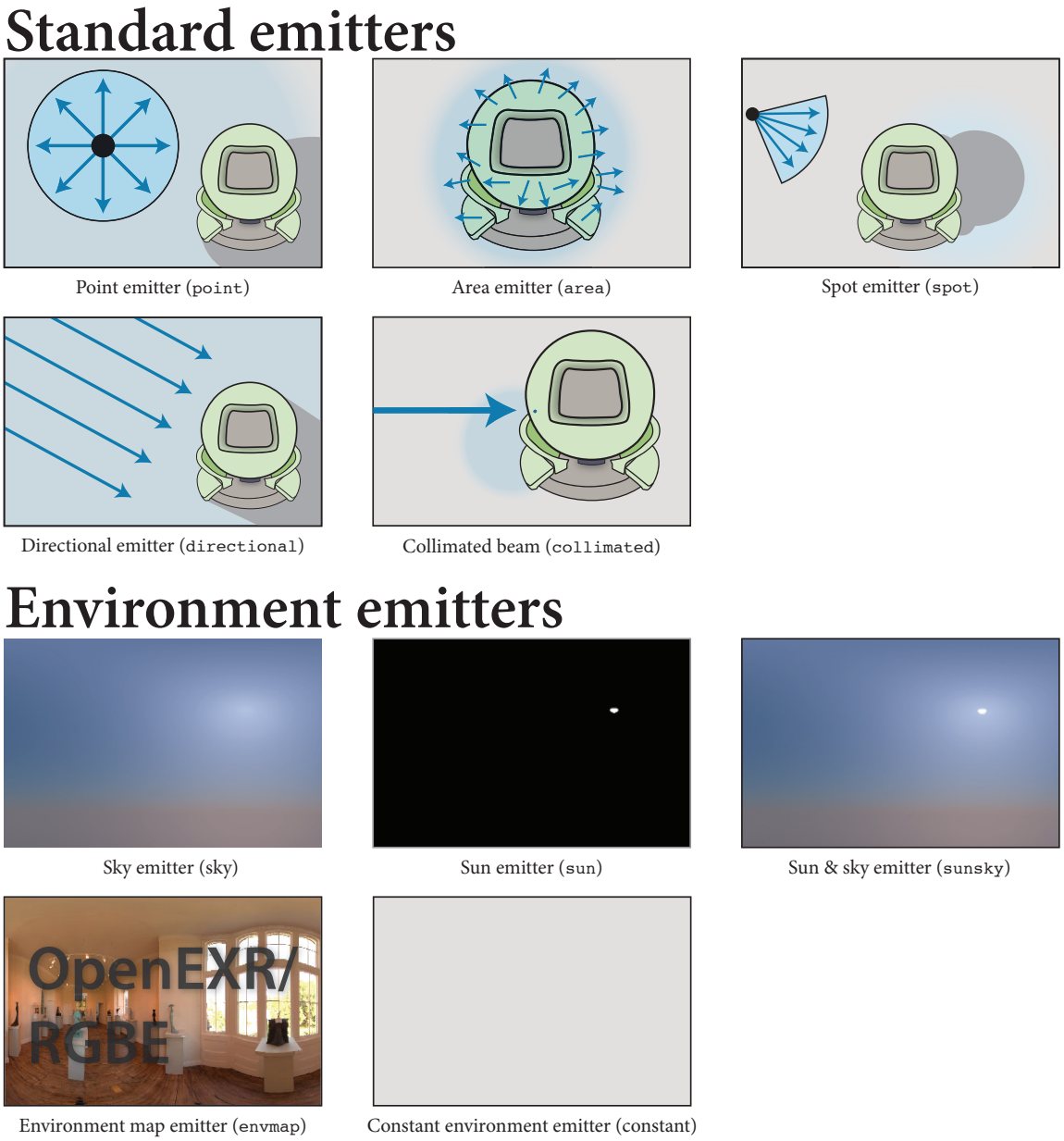


Figure 24: Schematic overview of the most important emitters in Mitsuba. The arrows indicate the directional distribution of light.

Generally, light sources are specified as children of the <scene> element; for instance, the following snippet instantiates a point light emitter that illuminates a sphere.

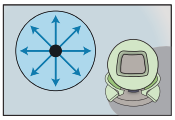
```
<scene version="0.5.0">
    <emitter type="point">
        <spectrum name="intensity" value="1"/>
        <point name="position" x="0" y="0" z="-2"/>
    </emitter>
    <shape type="sphere"/>
</scene>
```

An exception to this are *area lights*, which turn a geometric object into a light source. These are specified as children of the corresponding <shape> element.

```
<scene version="0.5.0">
    <shape type="sphere">
        <emitter type="area">
            <spectrum name="radiance" value="1"/>
        </emitter>
    </shape>
</scene>
```

Note the parameter names used to specify the light source power, which reflect the different associated physical units.

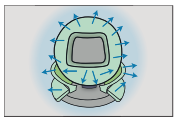
8.8.1. Point light source (`point`)



Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional sensor-to-world transformation. (Default: none (i.e. sensor space = world space))
`position`	`point`	Alternative parameter for specifying the light source position. Note that only one of the parameters `toWorld` and `position` can be used at a time.
`intensity`	`spectrum`	Specifies the radiant intensity in units of power per unit steradian. (Default: 1)
`samplingWeight`	`float`	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

This emitter plugin implements a simple point light source, which uniformly radiates illumination into all directions.

8.8.2. Area light (area)



Parameter	Type	Description
radiance	spectrum	Specifies the emitted radiance in units of power per unit area per unit steradian.
samplingWeight	float	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

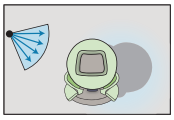
This plugin implements an area light, i.e. a light source that emits diffuse illumination from the exterior of an arbitrary shape. Since the emission profile of an area light is completely diffuse, it has the same apparent brightness regardless of the observer's viewing direction. Furthermore, since it occupies a nonzero amount of space, an area light generally causes scene objects to cast soft shadows.

When modeling scenes involving area lights, it is preferable to use spheres as the emitter shapes, since they provide a particularly good direct illumination sampling strategy (see the sphere plugin for an example).

To create an area light source, simply instantiate the desired emitter shape and specify an area instance as its child:

```
<!-- Create a spherical light source at the origin -->
<shape type="sphere">
    <emitter type="area">
        <spectrum name="radiance" value="1"/>
    </emitter>
</shape>
```

8.8.3. Spot light source (spot)



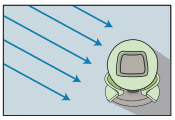
Parameter	Type	Description
toWorld	transform or animation	Specifies an optional sensor-to-world transformation. (Default: none (i.e. sensor space = world space))
intensity	spectrum	Specifies the maximum radiant intensity at the center in units of power per unit steradian. (Default: 1)
cutoffAngle	float	Cutoff angle, beyond which the spot light is completely black (Default: 20 degrees)
beamWidth	float	Subtended angle of the central beam portion (Default: cutoffAngle · 3/4)
texture	texture	An optional texture to be projected along the spot light
samplingWeight	float	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

This plugin provides a spot light with a linear falloff. In its local coordinate system, the spot light is positioned at the origin and points along the positive Z direction. It can be conveniently reoriented using the lookat tag, e.g.:

```
<emitter type="spot">
    <transform name="toWorld">
        <!-- Orient the light so that points from (1, 1, 1) towards (1, 2, 1) -->
        <lookat origin="1, 1, 1" target="1, 2, 1"/>
    </transform>
</emitter>
```

The intensity linearly ramps up from cutoffAngle to beamWidth (both specified in degrees), after which it remains at the maximum value. A projection texture may optionally be supplied.

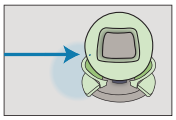
8.8.4. Directional emitter (`directional`)



Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional emitter-to-world transformation. (Default: none (i.e. emitter space = world space))
`direction`	`vector`	Alternative to `toWorld`: explicitly specifies the illumination direction. Note that only one of the two parameters can be used.
`irradiance`	`spectrum`	Specifies the amount of power per unit area received by a hypothetical surface normal to the specified direction (Default: 1)
`samplingWeight`	`float`	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

This emitter plugin implements a distant directional source, which radiates a specified power per unit area along a fixed direction. By default, the emitter radiates in the direction of the postive Z axis.

8.8.5. Collimated beam emitter (`collimated`)



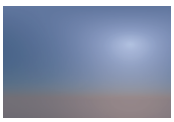
Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional emitter-to-world transformation. (Default: none (i.e. emitter space = world space))
`power`	`spectrum`	Specifies the amount of power radiated along the beam (Default: 1)
`samplingWeight`	`float`	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

This emitter plugin implements a collimated beam source, which radiates a specified amount of power along a fixed ray. It can be thought of as the limit of a spot light as its field of view tends to zero.

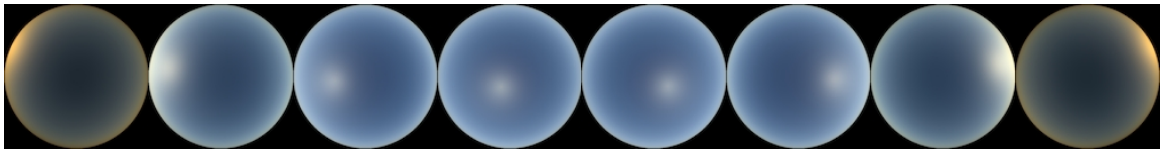
Such a emitter is useful for conducting virtual experiments and testing the renderer for correctness.

By default, the emitter is located at the origin and radiates into the positive Z direction $(0, 0, 1)$. This can be changed by providing a custom `toWorld` transformation.

8.8.6. Skylight emitter (sky)



Parameter	Type	Description
turbidity	float	This parameter determines the amount of aerosol present in the atmosphere. Valid range: 1-10. (Default: 3, corresponding to a clear sky in a temperate climate)
albedo	spectrum	Specifies the ground albedo (Default: 0.15)
year, month, day	integer	Denote the date of the observation (Default: 2010, 07, 10)
hour,minute,↙ second	float	Local time at the location of the observer in 24-hour format (Default: 15, 00, 00, i.e. 3PM)
latitude, longitude, timezone	float	These three parameters specify the oberver's latitude and longitude in degrees, and the local timezone offset in hours, which are required to compute the sun's position. (Default: 35.6894, 139.6917, 9 — Tokyo, Japan)
sunDirection	vector	Allows to manually override the sun direction in world space. When this value is provided, parameters pertaining to the computation of the sun direction (year, hour, latitude, etc. are unnecessary. (Default: none)
stretch	float	Stretch factor to extend emitter below the horizon, must be in [1,2] (Default: 1, i.e. not used)
resolution	integer	Specifies the horizontal resolution of the precomputed image that is used to represent the sun environment map (Default: 512, i.e. 512×256)
scale	float	This parameter can be used to scale the the amount of illumination emitted by the sky emitter. (Default: 1)
samplingWeight	float	Specifies the relative amount of samples allocated to this emitter. (Default: 1)
toWorld	transform or animation	Specifies an optional sensor-to-world transformation. (Default: none (i.e. sensor space = world space))



(a) 5AM (b) 7AM (c) 9AM (d) 11AM (e) 1PM (f) 3PM (g) 5PM (h) 6:30 PM

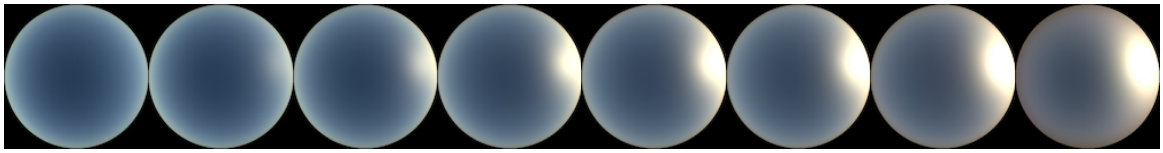
Figure 25: Time series at the default settings (Equidistant fisheye projection of the sky onto a disk. East is left.)

This plugin provides the physically-based skylight model by Hošek and Wilkie [31]. It can be used to create predictive daylight renderings of scenes under clear skies, which is useful for architectural and computer vision applications. The implementation in Mitsuba is based on code that was generously provided by the authors.

The model has two main parameters: the turbidity of the atmosphere and the position of the

sun. The position of the sun in turn depends on a number of secondary parameters, including the `latitude`, `longitude`, and `timezone` at the location of the observer, as well as the current `year`, `month`, `day`, `hour`, `minute`, and `second`. Using all of these, the elevation and azimuth of the sun are computed using the PSA algorithm by Blanco et al. [2], which is accurate to about 0.5 arcminutes (1/120 degrees). Note that this algorithm does not account for daylight savings time where it is used, hence a manual correction of the time may be necessary. For detailed coordinate and timezone information of various cities, see http://www.esrl.noaa.gov/gmd/grad/solcalc.

If desired, the world-space solar vector may also be specified using the `sunDirection` parameter, in which case all of the previously mentioned time and location parameters become irrelevant.



(a) 1 (b) 2 (c) 3 (d) 4 (e) 5 (f) 6 (g) 8 (h) 10

Figure 26: Sky light for different turbidity values (default configuration at 5PM)

Turbidity, the other important parameter, specifies the aerosol content of the atmosphere. Aerosol particles cause additional scattering that manifests in a halo around the sun, as well as color fringes near the horizon. Smaller turbidity values ($\sim 1-2$) produce an arctic-like clear blue sky, whereas larger values ($\sim 8-10$) create an atmosphere that is more typical of a warm, humid day. Note that this model does not aim to reproduce overcast, cloudy, or foggy atmospheres with high corresponding turbidity values. An photographic environment map may be more appropriate in such cases.

The default coordinate system of the emitter associates the up direction with the $+Y$ axis. The east direction is associated with $+X$ and the north direction is equal to $+Z$. To change this coordinate system, rotations can be applied using the `toWorld` parameter (see Listing 34 for an example).

By default, the emitter will not emit any light below the horizon, which means that these regions are black when observed directly. By setting the `stretch` parameter to values between 1 and 2, the sky can be extended to cover these directions as well. This is of course a complete kludge and only meant as a quick workaround for scenes that are not properly set up.

Instead of evaluating the full sky model every on every radiance query, the implementation pre-computes a low resolution environment map (512× 256) of the entire sky that is then forwarded to the `envmap` plugin—this dramatically improves rendering performance. This resolution is generally plenty since the sky radiance distribution is so smooth, but it it can be adjusted manually if necessary using the `resolution` parameter.

Note that while the model encompasses sunrise and sunset configurations, it does not extend to the night sky, where illumination from stars, galaxies, and the moon dominate. When started with a sun configuration that lies below the horizon, the plugin will fail with an error message.

```
<emitter type="sky">
    <transform name="toWorld">
        <rotate x="1" angle="90"/>
    </transform>
</emitter>
```

Listing 34: Rotating the sky emitter for scenes that use Z as the "up" direction

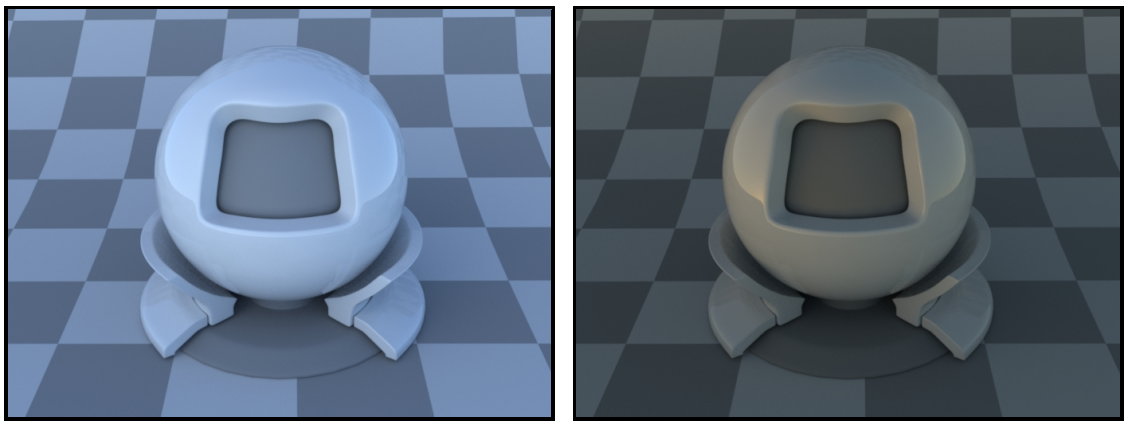
Physical units and spectral rendering

Like the `blackbody` emission profile (Page 26), the sky model introduces physical units into the rendering process. The radiance values computed by this plugin have units of power (W) per unit area (m^{-2}) per steradian (sr^{-1}) per unit wavelength (nm^{-1}). If these units are inconsistent with your scene description, you may use the optional `scale` parameter to adjust them.

When Mitsuba is compiled for spectral rendering, the plugin switches from RGB to a spectral variant of the skylight model, which relies on precomputed data between 320 and 720nm sampled at 40nm-increments.

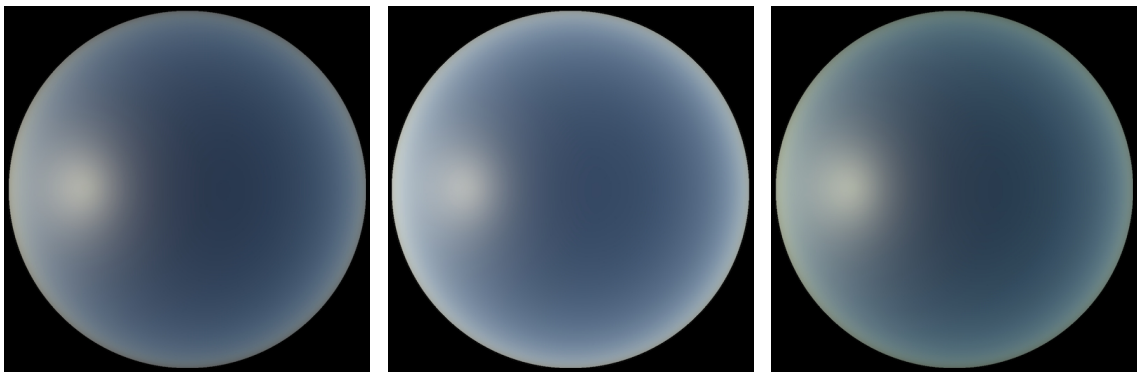
Ground albedo

The albedo of the ground (e.g. due to rock, snow, or vegetation) can have a noticeable and nonlinear effect on the appearance of the sky. Figure 28 shows an example of this effect. By default, the ground albedo is set to a 15% gray.



(a) 3 PM (b) 6:30 PM

Figure 27: Renderings with the `plastic` material under default conditions. Note that these only contain skylight illumination. For a model that also includes the sun, refer to `sunsky`.



(a) `albedo=0%` (b) `albedo=100%` (c) `albedo=20% green`

Figure 28: Influence of the ground albedo on the apperance of the sky

8.8.7. Sun emitter (sun)

Parameter	Type	Description
turbidity	float	This parameter determines the amount of aerosol present in the atmosphere. Valid range: 2-10. (Default: 3, corresponding to a clear sky in a temperate climate)
year, month, day	integer	Denote the date of the observation (Default: 2010, 07, 10)
hour,minute,↙ second	float	Local time at the location of the observer in 24-hour format (Default: 15, 00, 00, i.e. 3PM)
latitude, longitude, timezone	float	These three parameters specify the oberver's latitude and longitude in degrees, and the local timezone offset in hours, which are required to compute the sun's position. (Default: 35.6894, 139.6917, 9 — Tokyo, Japan)
sunDirection	vector	Allows to manually override the sun direction in world space. When this value is provided, parameters pertaining to the computation of the sun direction (year, hour, latitude, etc. are unnecessary. (Default: none)
resolution	integer	Specifies the horizontal resolution of the precomputed image that is used to represent the sun environment map (Default: 512, i.e. 512×256)
scale	float	This parameter can be used to scale the the amount of illumination emitted by the sun emitter. (Default: 1)
sunRadiusScale	float	Scale factor to adjust the radius of the sun, while preserving its power. Set to 0 to turn it into a directional light source.
samplingWeight	float	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

This plugin implements the physically-based sun model proposed by Preetham et al. [38]. Using the provided position and time information (see sky for details), it can determine the position of the sun as seen from the position of the observer. The radiance arriving at the earth surface is then found based on the spectral emission profile of the sun and the extinction cross-section of the atmosphere (which depends on the turbidity and the zenith angle of the sun).

Like the blackbody emission profile (Page 26), the sun model introduces physical units into the rendering process. The radiance values computed by this plugin have units of power (W) per unit area (m^{-2}) per steradian (sr^{-1}) per unit wavelength (nm^{-1}). If these units are inconsistent with your scene description, you may use the optional scale parameter to adjust them.

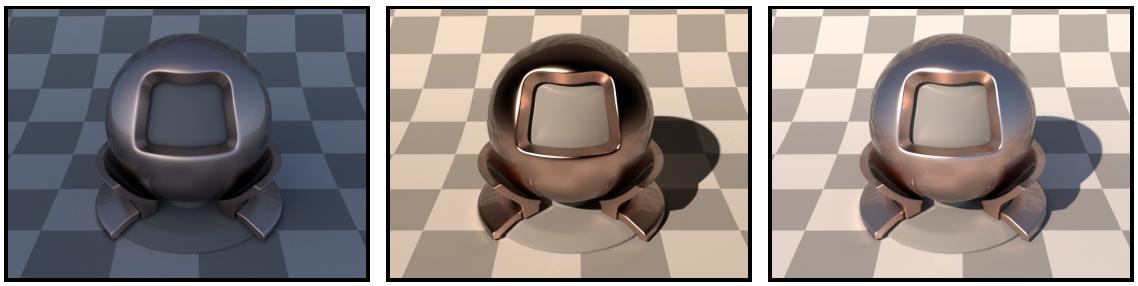
This plugin supplies proper spectral power distributions when Mitsuba is compiled in spectral rendering mode. Otherwise, they are simply projected onto a linear RGB color space.

Remarks:

- The sun is an intense light source that subtends a tiny solid angle. This can be a problem for certain rendering techniques (e.g. path tracing), which produce high variance output (i.e. noise in renderings) when the scene also contains specular or glossy or materials.

8.8.8. Sun and sky emitter (sunsky)

Parameter	Type	Description
turbidity	float	This parameter determines the amount of aerosol present in the atmosphere. Valid range: 1-10. (Default: 3, corresponding to a clear sky in a temperate climate)
albedo	spectrum	Specifies the ground albedo (Default: 0.15)
year, month, day	integer	Denote the date of the observation (Default: 2010, 07, 10)
hour,minute,↙ second	float	Local time at the location of the observer in 24-hour format (Default: 15, 00, 00, i.e. 3PM)
latitude, longitude, timezone	float	These three parameters specify the oberver's latitude and longitude in degrees, and the local timezone offset in hours, which are required to compute the sun's position. (Default: 35.6894, 139.6917, 9 — Tokyo, Japan)
sunDirection	vector	Allows to manually override the sun direction in world space. When this value is provided, parameters pertaining to the computation of the sun direction (year, hour, latitude, etc. are unnecessary. (Default: none)
stretch	float	Stretch factor to extend emitter below the horizon, must be in [1,2] (Default: 1, i.e. not used)
resolution	integer	Specifies the horizontal resolution of the precomputed image that is used to represent the sun environment map (Default: 512, i.e. 512×256)
sunScale	float	This parameter can be used to separately scale the the amount of illumination emitted by the sun. (Default: 1)
skyScale	float	This parameter can be used to separately scale the the amount of illumination emitted by the sky. (Default: 1)
sunRadiusScale	float	Scale factor to adjust the radius of the sun, while preserving its power. Set to 0 to turn it into a directional light source.

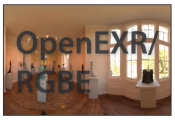


(a) sky emitter (b) sun emitter (c) sunsky emitter

Figure 29: A coated rough copper test ball lit with the three provided daylight illumination models

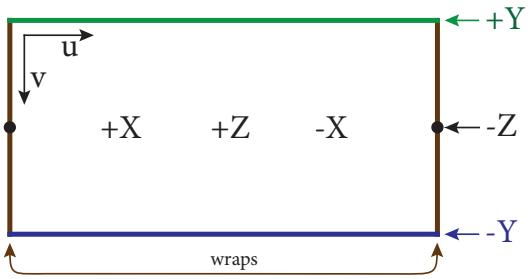
This convenience plugin has the sole purpose of instantiating sun and sky and merging them into a joint environment map. Please refer to these plugins individually for more details.

8.8.9. Environment emitter (envmap)



Parameter	Type	Description
filename	string	Filename of the radiance-valued input image to be loaded; must be in latitude-longitude format.
scale	float	A scale factor that is applied to the radiance values stored in the input image. (Default: 1)
toWorld	transform	Specifies an optional linear emitter-to-world space rotation. (Default: none (i.e. emitter space = world space))
gamma	float	Optional parameter to override the gamma value of the source bitmap, where 1 indicates a linear color space and the special value -1 corresponds to sRGB. (Default: automatically detect based on the image type and metadata)
cache	boolean	Preserve generated MIP map data in a cache file? This will cause a file named *filename*.mip to be created. (Default: automatic—use caching for images larger than 1M pixels.)
samplingWeight	float	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

(a) The museum environment map by Bernhard Vogl that is used in many example renderings in this document



(b) Coordinate conventions used when mapping the input image onto the sphere.

This plugin provides a HDRI (high dynamic range imaging) environment map, which is a type of light source that is well-suited for representing "natural" illumination. Many images in this document are made using the environment map shown in (a).

The implementation loads a captured illumination environment from a image in latitude-longitude format and turns it into an infinitely distant emitter. The image could either be be a processed photograph or a rendering made using the spherical sensor. The direction conventions of this transformation are shown in (b). The plugin can work with all types of images that are natively supported by Mitsuba (i.e. JPEG, PNG, OpenEXR, RGBE, TGA, and BMP). In practice, a good environment map will contain high-dynamic range data that can only be represented using the OpenEXR or RGBE file formats. High quality free light probes are available on Paul Debevec's website (http://gl.ict.usc.edu/Data/HighResProbes) and Bernhard Vogl's website (http://dativ.at/lightprobes/).

Like the bitmap texture, this plugin generates a cache file named *filename*.mip when given a large input image. This significantly accelerates the loading times of subsequent renderings. When this is not desired, specify cache=false to the plugin.

8.8.10. Constant environment emitter (`constant`)

Parameter	Type	Description
`radiance`	`spectrum`	Specifies the emitted radiance in units of power per unit area per unit steradian.
`samplingWeight`	`float`	Specifies the relative amount of samples allocated to this emitter. (Default: 1)

This plugin implements a constant environment emitter, which surrounds the scene and radiates diffuse illumination towards it. This is often a good default light source when the goal is to visualize some loaded geometry that uses basic (e.g. diffuse) materials.

8.9. Sensors

In Mitsuba, *sensors*, along with a *film*, are responsible for recording radiance measurements in some usable format. This includes default choices such as perspective or orthographic cameras, as well as more specialized sensors that measure the radiance into a given direction or the irradiance received by a certain surface. This subsection lists the available choices.

Syntax

In the XML scene description language, a sensor declaration looks as follows

```
<scene version="0.5.0">
    <!-- ... scene contents ... -->

    <sensor type="... sensor type ...">
        <!-- ... sensor parameters ... -->

        <sampler type=" ... sampler type ... ">
            <!-- ... sampler parameters ... -->
        </sampler>

        <film type=" ... film type ... ">
            <!-- ... film parameters ... -->
        </film>
    </sensor>
</scene>
```

In other words, the `<sensor>` declaration is a child element of the `<scene>` (the particular position in the scene file does not play a role). Nested within the sensor declaration is a sampler instance (described in Section 8.11) and a film instance (described in Section 8.12).

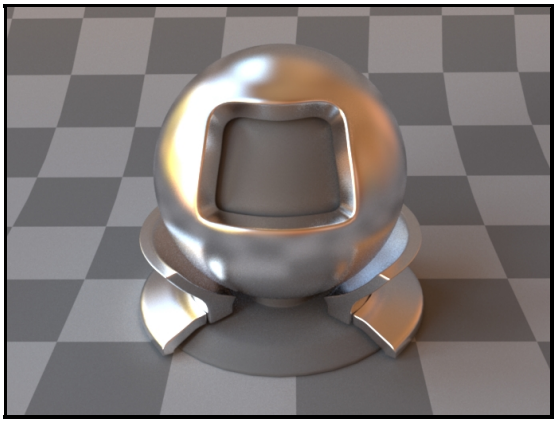
Handedness convention

Sensors in Mitsuba are *right-handed*. Any number of rotations and translations can be applied to them without changing this property. By default they are located at the origin and oriented in such a way that in the rendered image, $+X$ points left, $+Y$ points upwards, and $+Z$ points along the viewing direction.

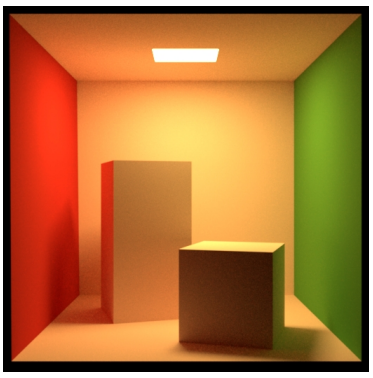
Left-handed sensors are also supported. To switch the handedness, flip any one of the axes, e.g. by passing a scale transformation like `<scale x="-1"/>` to the sensor's `toWorld` parameter.

8.9.1. Perspective pinhole camera (`perspective`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional camera-to-world transformation. (Default: none (i.e. camera space = world space))
`focalLength`	`string`	Denotes the camera's focal length specified using `35mm` film equivalent units. See the main description for further details. (Default: `50mm`)
`fov`	`float`	An alternative to `focalLength`: denotes the camera's field of view in degrees—must be between 0 and 180, excluding the extremes.
`fovAxis`	`string`	When the parameter `fov` is given (and only then), this parameter further specifies the image axis, to which it applies. (i) **`x`**: `fov` maps to the `x`-axis in screen space. (ii) **`y`**: `fov` maps to the `y`-axis in screen space. (iii) **`diagonal`**: `fov` maps to the screen diagonal. (iv) **`smaller`**: `fov` maps to the smaller dimension (e.g. `x` when `width<height`) (v) **`larger`**: `fov` maps to the larger dimension (e.g. `y` when `width<height`) The default is **`x`**.
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)
`nearClip`, `farClip`	`float`	Distance to the near/far clip planes. (Default: `nearClip=1e-2` (i.e. `0.01`) and `farClip=1e4` (i.e. `10000`))



(a) The material test ball viewed through a perspective pinhole camera. Everything is in sharp focus.



(b) A rendering of the Cornell box

This plugin implements a simple idealizied perspective camera model, which has an infinitely small aperture. This creates an infinite depth of field, i.e. no optical blurring occurs. The camera is can be

specified to move during an exposure, hence temporal blur is still possible.

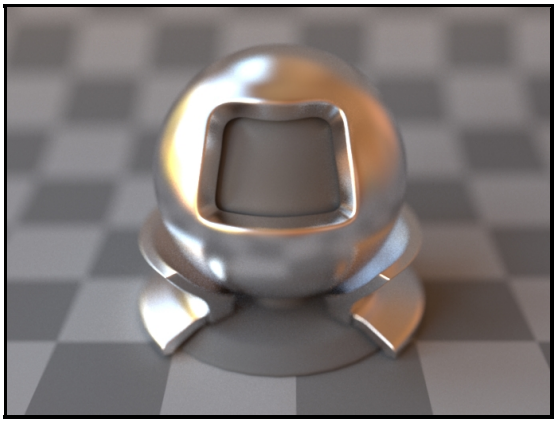
By default, the camera's field of view is specified using a 35mm film equivalent focal length, which is first converted into a diagonal field of view and subsequently applied to the camera. This assumes that the film's aspect ratio matches that of 35mm film (1.5:1), though the parameter still behaves intuitively when this is not the case. Alternatively, it is also possible to specify a field of view in degrees along a given axis (see the `fov` and `fovAxis` parameters).

The exact camera position and orientation is most easily expressed using the `lookat` tag, i.e.:

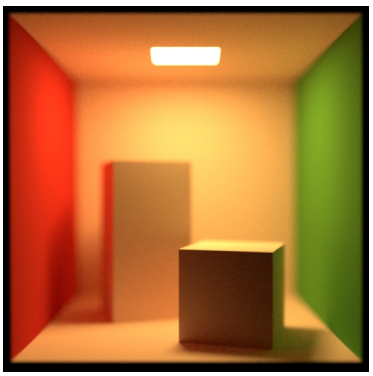
```
<sensor type="perspective">
    <transform name="toWorld">
        <!-- Move and rotate the camera so that looks from (1, 1, 1) to (1, 2, 1)
             and the direction (0, 0, 1) points "up" in the output image -->
        <lookat origin="1, 1, 1" target="1, 2, 1" up="0, 0, 1"/>
    </transform>
</sensor>
```

8.9.2. Perspective camera with a thin lens (`thinlens`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional camera-to-world transformation. (Default: none (i.e. camera space = world space))
`apertureRadius`	`float`	Denotes the radius of the camera's aperture in scene units.
`focusDistance`	`float`	Denotes the world-space distance from the camera's aperture to the focal plane. (Default: `0`)
`focalLength`	`string`	Denotes the camera's focal length specified using `35mm` film equivalent units. See the main description for further details. (Default: `50mm`)
`fov`	`float`	An alternative to `focalLength`: denotes the camera's field of view in degrees—must be between 0 and 180, excluding the extremes.
`fovAxis`	`string`	When the parameter `fov` is given (and only then), this parameter further specifies the image axis, to which it applies. (i) **`x`**: `fov` maps to the `x`-axis in screen space. (ii) **`y`**: `fov` maps to the `y`-axis in screen space. (iii) **`diagonal`**: `fov` maps to the screen diagonal. (iv) **`smaller`**: `fov` maps to the smaller dimension (e.g. `x` when `width<height`) (v) **`larger`**: `fov` maps to the larger dimension (e.g. `y` when `width<height`) The default is **`x`**.
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)
`nearClip`, `farClip`	`float`	Distance to the near/far clip planes. (Default: `nearClip=1e-2` (i.e. `0.01`) and `farClip=1e4` (i.e. `10000`))



(a) The material test ball viewed through a perspective thin lens camera. Points away from the focal plane project onto a circle of confusion.



(b) A rendering of the Cornell box

This plugin implements a simple perspective camera model with a thin lens at its circular aperture. It is very similar to the `perspective` plugin except that the extra lens element permits rendering with a specifiable (i.e. non-infinite) depth of field. To configure this, it has two extra parameters named `apertureRadius` and `focusDistance`.

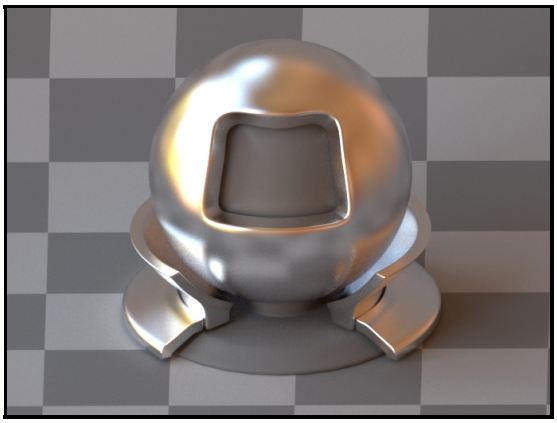
By default, the camera's field of view is specified using a 35mm film equivalent focal length, which is first converted into a diagonal field of view and subsequently applied to the camera. This assumes that the film's aspect ratio matches that of 35mm film (1.5:1), though the parameter still behaves intuitively when this is not the case. Alternatively, it is also possible to specify a field of view in degrees along a given axis (see the `fov` and `fovAxis` parameters).

The exact camera position and orientation is most easily expressed using the `lookat` tag, i.e.:

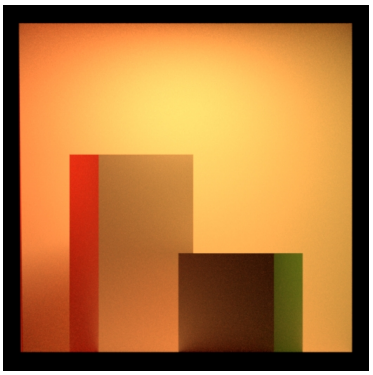
```
<sensor type="thinlens">
    <transform name="toWorld">
        <!-- Move and rotate the camera so that looks from (1, 1, 1) to (1, 2, 1)
             and the direction (0, 0, 1) points "up" in the output image -->
        <lookat origin="1, 1, 1" target="1, 2, 1" up="0, 0, 1"/>
    </transform>

    <!-- Focus on the target -->
    <float name="focusDistance" value="1"/>
    <float name="apertureRadius" value="0.1"/>
</sensor>
```

8.9.3. Orthographic camera (`orthographic`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional camera-to-world transformation. (Default: none (i.e. camera space = world space))
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)
`nearClip`, `farClip`	`float`	Distance to the near/far clip planes. (Default: `nearClip=1e-2` (i.e. `0.01`) and `farClip=1e4` (i.e. `10000`))



(a) The material test ball viewed through an orthographic camera. Note the complete lack of perspective.



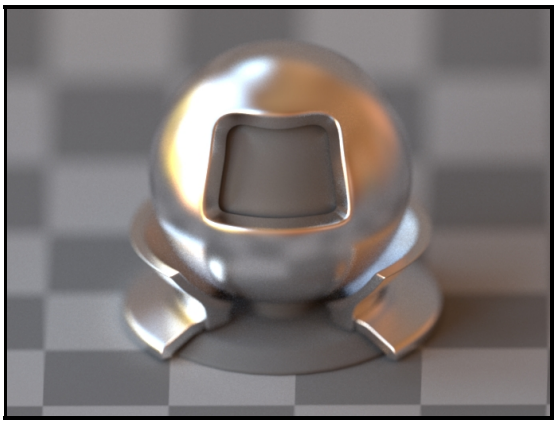
(b) A rendering of the Cornell box

This plugin implements a simple orthographic camera, i.e. a sensor based on an orthographic projection without any form of perspective. It can be thought of as a planar sensor that measures the radiance along its normal direction. By default, this is the region $[-1,1]^2$ inside the XY-plane facing along the positive Z direction. Transformed versions can be instantiated e.g. as follows:

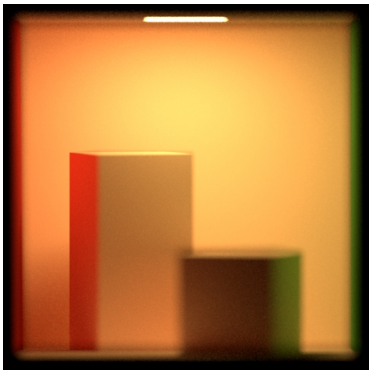
```
<sensor type="orthographic">
    <transform name="toWorld">
        <!-- Resize the sensor plane to 20x20 world space units -->
        <scale x="10" y="10"/>

        <!-- Move and rotate it so that it contains the point
             (1, 1, 1) and faces direction (0, 1, 0) -->
        <lookat origin="1, 1, 1" target="1, 2, 1" up="0, 0, 1"/>
    </transform>
</sensor>
```

8.9.4. Telecentric lens camera (`telecentric`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional sensor-to-world transformation. (Default: none (i.e. camera space = world space))
`apertureRadius`	`float`	Denotes the radius of the camera's aperture in scene units. (Default: `0`)
`focusDistance`	`float`	Denotes the world-space distance from the camera's aperture to the focal plane. (Default: `0`)
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)
`nearClip`, `farClip`	`float`	Distance to the near/far clip planes. (Default: `nearClip=1e-2` (i.e. `0.01`) and `farClip=1e4` (i.e. `10000`))



(a) The material test ball viewed through an telecentric camera. Note the orthographic view together with a narrow depth of field.



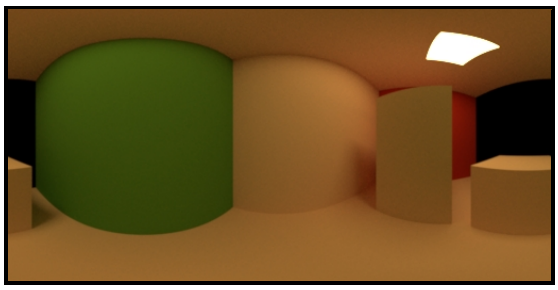
(b) A rendering of the Cornell box. The red and green walls are partially visible due to the aperture size.

This plugin implements a simple model of a camera with a *telecentric lens*. This is a type of lens that produces an in-focus orthographic view on a plane at some distance from the sensor. Points away from this plane are out of focus and project onto a circle of confusion. In comparison to idealized orthographic cameras, telecentric lens cameras exist in the real world and find use in some computer vision applications where perspective effects cause problems. This sensor relates to the `orthographic` plugin in the same way that `thinlens` does to `perspective`.

The configuration is identical to the `orthographic` plugin, except that the additional parameters `apertureRadius` and `focusDistance` must be provided.

8.9.5. Spherical camera (`spherical`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional camera-to-world transformation. (Default: none (i.e. camera space = world space))
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)



(a) A rendering made using a spherical camera

The spherical camera captures the illumination arriving from all directions and turns it into a latitude-longitude environment map. It is best used with a high dynamic range film that has 2:1 aspect ratio, and the resulting output can then be turned into a distant light source using the `envmap` plugin. By default, the camera is located at the origin, which can be changed by providing a custom `toWorld` transformation.

8.9.6. Irradiance meter (`irradiancemeter`)

Parameter	Type	Description
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)

This sensor plugin implements a simple irradiance meter, which measures the average incident power per unit area over a provided surface. Such a sensor is useful for conducting virtual experiments and testing the renderer for correctness. The result is normalized so that an irradiance sensor inside an integrating sphere with constant radiance 1 records an irradiance value of π.

To create an irradiance meter, instantiate the desired measurement shape and specify the sensor as its child. Note that when the sensor's film resolution is larger than 1×1, each pixel will record the average irradiance over a rectangular part of the shape's UV parameterization.

```
<scene version="0.5.0">
    <!-- Measure the average irradiance arriving on
         a unit radius sphere located at the origin -->
    <shape type="sphere">
        <sensor type="irradiancemeter">
            <!-- Write the output to a MATLAB M-file. The output file will
                 contain a 1x1 matrix storing an estimate of the average
                 irradiance over the surface of the sphere. -->
            <film type="mfilm"/>

            <!-- Use 1024 samples for the measurement -->
            <sampler type="independent">
                <integer name="sampleCount" value="1024"/>
            </sampler>
        </sensor>
    </shape>

    <!-- ... other scene declarations ... -->
</scene>
```

8.9.7. Radiance meter (radiancemeter)

Parameter	Type	Description
toWorld	transform or animation	Specifies an optional sensor-to-world transformation. (Default: none (i.e. sensor space = world space))
shutterOpen, shutterClose	float	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)

This sensor plugin implements a simple radiance meter, which measures the incident power per unit area per unit solid angle along a certain ray. It can be thought of as the limit of a standard perspective camera as its field of view tends to zero. Hence, when this sensor is given a film with multiple pixels, all of them will record the same value.

Such a sensor is useful for conducting virtual experiments and testing the renderer for correctness.

By default, the sensor is located at the origin and performs a measurement in the positive Z direction $(0, 0, 1)$. This can be changed by providing a custom `toWorld` transformation:

```
<scene version="0.5.0">
    <sensor type="radiancemeter">
        <!-- Measure the amount of radiance traveling
             from the origin to (1,2,3) -->
        <transform name="toWorld">
            <lookat origin="1,2,3"
                    target="0,0,0"/>
        </transform>

        <!-- Write the output to a MATLAB M-file. The output file will
             contain a 1x1 matrix storing an estimate of the incident
             radiance along the specified ray. -->
        <film type="mfilm"/>

        <!-- Use 1024 samples for the measurement -->
        <sampler type="independent">
            <integer name="sampleCount" value="1024"/>
        </sampler>
    </sensor>

    <!-- ... other scene declarations ... -->
</scene>
```

8.9.8. Fluence meter (`fluencemeter`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional sensor-to-world transformation. (Default: none (i.e. sensor space = world space))
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)

This sensor plugin implements a simple fluence meter, which measures the average radiance passing through a specified position. By default, the sensor is located at the origin.

Such a sensor is useful for conducting virtual experiments and testing the renderer for correctness.

```
<scene version="0.5.0">
    <sensor type="fluencemeter">
        <!-- Measure the average radiance traveling
             through the point (1,2,3) -->
        <transform name="toWorld">
            <translate x="1" y="2" z="3"/>
        </transform>

        <!-- Write the output to a MATLAB M-file. The output file will
             contain a 1x1 matrix storing the computed estimate -->
        <film type="mfilm"/>

        <!-- Use 1024 samples for the measurement -->
        <sampler type="independent">
            <integer name="sampleCount" value="1024"/>
        </sampler>
    </sensor>

    <!-- ... other scene declarations ... -->
</scene>
```

8.9.9. Perspective pinhole camera with radial distortion (`perspective_rdist`)

Parameter	Type	Description
`toWorld`	`transform` or `animation`	Specifies an optional camera-to-world transformation. (Default: none (i.e. camera space = world space))
`kc`	`string`	Second and fourth-order coefficient of a polynomial radial distortion model specified as a comma-separated list The specifics of the model are described in detail on the following page: http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/parameters.html
`focalLength`	`string`	Denotes the camera's focal length specified using `35mm` film equivalent units. See the main description for further details. (Default: `50mm`)
`fov`	`float`	An alternative to `focalLength`: denotes the camera's field of view in degrees—must be between 0 and 180, excluding the extremes.
`fovAxis`	`string`	When the parameter `fov` is given (and only then), this parameter further specifies the image axis, to which it applies. (i) **`x`**: `fov` maps to the `x`-axis in screen space. (ii) **`y`**: `fov` maps to the `y`-axis in screen space. (iii) **`diagonal`**: `fov` maps to the screen diagonal. (iv) **`smaller`**: `fov` maps to the smaller dimension (e.g. `x` when `width<height`) (v) **`larger`**: `fov` maps to the larger dimension (e.g. `y` when `width<height`) The default is **`x`**.
`shutterOpen`, `shutterClose`	`float`	Specifies the time interval of the measurement—this is only relevant when the scene is in motion. (Default: 0)
`nearClip`, `farClip`	`float`	Distance to the near/far clip planes. (Default: `nearClip=1e-2` (i.e. `0.01`) and `farClip=1e4` (i.e. `10000`))

This plugin extends the `perspective` camera with support for radial distortion. It accepts an additional parameter named `kc`, which specifies the second and fourth-order terms in a polynomial model that accounts for pincushion and barrel distortion. This is useful when trying to match rendered images to photographs created by a camera whose distortion is known. When `kc=0, 0`, the model turns into a standard pinhole camera. The reason for creating a separate plugin for this feature is that distortion involves extra overheads per ray that users may not be willing to pay for if their scene doesn't use it. The MATLAB Camera Calibration Toolbox by Jean-Yves Bouguet (http://www.vision.caltech.edu/bouguetj/calib_doc/) can be used to obtain a distortion model, and the first entries of the `kc` variable generated by this tool can directly be passed into this plugin.

8.10. Integrators

In Mitsuba, the different rendering techniques are collectively referred to as *integrators*, since they perform integration over a high-dimensional space. Each integrator represents a specific approach for solving the light transport equation—usually favored in certain scenarios, but at the same time affected by its own set of intrinsic limitations. Therefore, it is important to carefully select an integrator based on user-specified accuracy requirements and properties of the scene to be rendered.

In Mitsuba's XML description language, a single integrator is usually instantiated by declaring it at the top level within the scene, e.g.

```
<scene version="0.5.0">
    <!-- Instantiate a unidirectional path tracer,
         which renders paths up to a depth of 5 -->
    <integrator type="path">
        <integer name="maxDepth" value="5"/>
    </integrator>

    <!-- Some geometry to be rendered -->
    <shape type="sphere">
        <bsdf type="diffuse"/>
    </shape>
</scene>
```

This section gives a brief overview of the available choices along with their parameters.

Choosing an integrator

Due to the large number of integrators in Mitsuba, the decision of which one is suitable may seem daunting. Assuming that the goal is to solve the full light transport equation without approximations, a few integrators (`ao`, `direct`, `vpl`) can already be ruled out. The adjoint particle tracer `ptracer` is also rarely used.

The following "algorithm" may help to decide amongst the remaining ones:

1. Try rendering the scene with an appropriate path tracer. If this gives the desired result, stop.

   Mitsuba currently comes with three path tracer variations that target different setups: If your scene contains no media and no surfaces with opacity masks, use the plain path tracer (`path`).

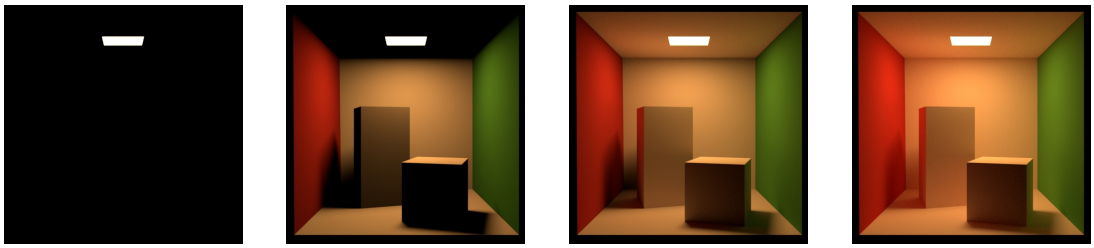
   Otherwise, use one of the volumetric path tracers (`volpath_simple` or `volpath`). The latter is preferable if the scene contains glossy surface scattering models.

2. If step 1 produced poor (i.e. noisy and slowly converging) results, try the bidirectional path tracer (`bdpt`).

3. If steps 1 and 2 failed, the scene contains a relatively difficult lighting setup, potentially including interaction with complex materials. In many cases, these difficulties can be greatly ameliorated by running a "metropolized" version of a path tracer. This is implemented in the Primary Sample Space MLT (`pssmlt`) plugin.

4. If none of the above worked, the remaining options are to try a photon mapping-type method (`photonmapper`, `ppm`, `sppm`) or a path-space MLT method (`mlt`, `erpt`).

Path depth



(a) Max. depth = 1 (b) Max. depth = 2 (c) Max. depth = 3 (d) Max. depth = ∞

Figure 30: These Cornell box renderings demonstrate the visual effect of a maximum path depth. As the paths are allowed to grow longer, the color saturation increases due to multiple scattering interactions with the colored surfaces. At the same time, the computation time increases.

Almost all integrators use the concept of *path depth*. Here, a path refers to a chain of scattering events that starts at the light source and ends at the eye or sensor. It is often useful to limit the path depth (Figure 30) when rendering scenes for preview purposes, since this reduces the amount of computation that is necessary per pixel. Furthermore, such renderings usually converge faster and therefore need fewer samples per pixel. When reference-quality is desired, one should always leave the path depth unlimited.

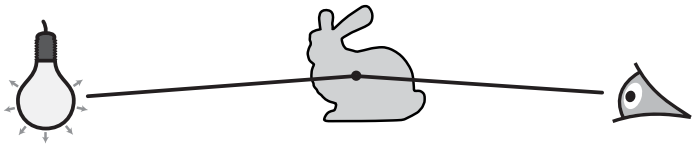


Figure 31: A ray of emitted light is scattered by an object and subsequently reaches the eye/sensor. In Mitsuba, this is a *depth-2* path, since it has two edges.

Mitsuba counts depths starting at 1, which correspond to visible light sources (i.e. a path that starts at the light source and ends at the eye or sensor without any scattering interaction in between). A depth-2 path (also known as "direct illumination") includes a single scattering event (Figure 31).

Progressive versus non-progressive

Some of the rendering techniques in Mitsuba are *progressive*. What this means is that they display a rough preview, which improves over time. Leaving them running indefinitely will continually reduce noise (in unbiased algorithms such as Metropolis Light Transport) or noise and bias (in biased rendering techniques such as Progressive Photon Mapping).

Hiding directly visible emitters

Several rendering algorithms in Mitsuba have a feature to hide directly visible light sources (e.g. environment maps or area lights). While not particularly realistic, this feature is often convenient to remove a background from a rendering so that it can be pasted into a differently-colored document.

Note that only directly visible emitters can be hidden using this feature—a reflection on a shiny surface will be unaffected. To perform the kind of compositing shown in Figure 32, it is also necessary to enable the alpha channel in the scene's film instance (Section 8.12).

(a) Daylit smoke rendered with `hideEmitters` set to `false` (the default setting)

(b) Rendered with `hideEmitters` set to `true` and alpha-composited onto a white background.

Figure 32: An example application of the `hideEmitters` parameter together with alpha blending

Number of samples per pixel

Many of the integrators in Mitsuba depend on a number of *samples per pixel*, which is related to the amount of noise in the final output. However, it is important to note that this parameter is *not* a parameter of the integrator. Instead, it must be declared in the <sampler> instantiation, which is nested inside the <sampler> element. The rationale behind this is that the sensor is responsible for performing a measurement using a specified sampling strategy. The integrator declares the high-level strategy for resolving scattering interactions, but does not depend on a specific number of samples.

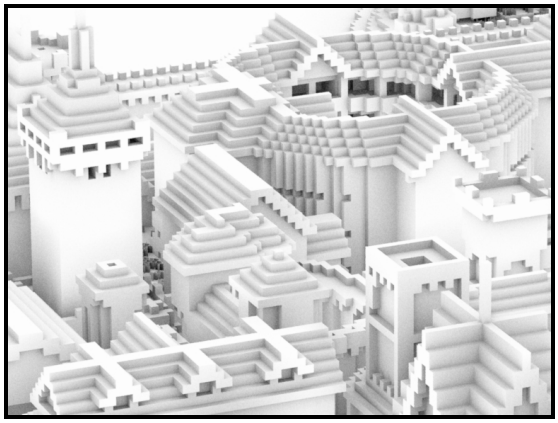
```
<scene version="0.5.0">
    <!-- ... scene contents ... -->

    <integrator type=".. integrator type ..">
        <!-- ... integrator parameters ... -->
    </integrator>

    <sensor type="... sensor type ...">
        <!-- ... sensor parameters ... -->

        <sampler type=" ... sampler type ... ">
            <!-- ... sampler parameters ... -->

            <!-- Important: number of samples per pixel goes here -->
            <integer name="sampleCount" value="32"/>
```

```
        </sampler>

        <film type=" ... film type ... ">
            <!-- ... film parameters ... -->
        </film>
    </sensor>
</scene>
```

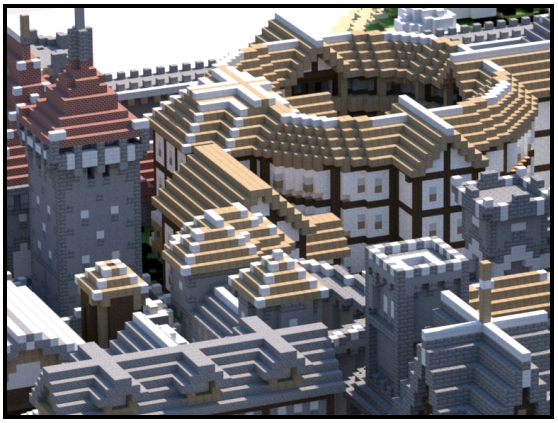
8.10.1. Ambient occlusion integrator (`ao`)

Parameter	Type	Description
`shadingSamples`	`integer`	Specifies the number of shading samples that should be computed per primary ray (Default: 1)
`rayLength`	`float`	Specifies the world-space length of the ambient occlusion rays that will be cast. (Default: -1, i.e. automatic)

.



(a) A view of the scene on page 41, rendered using the Ambient Occlusion integrator

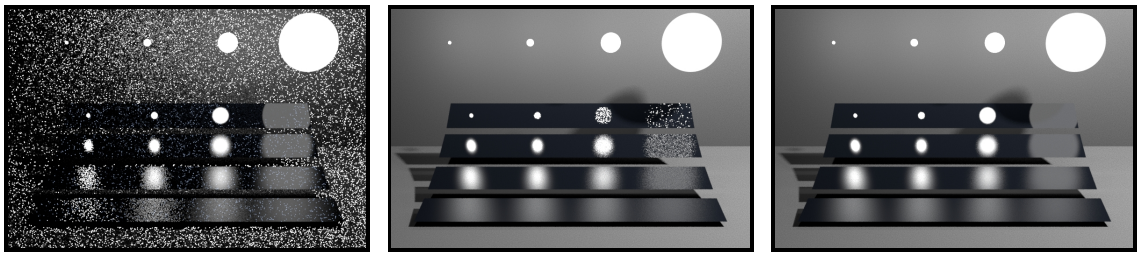


(b) A corresponding rendering created using the standard `path` tracer

Ambient Occlusion is a simple non-photorealistic rendering technique that simulates the exposure of an object to uniform illumination incident from all direction. It produces approximate shadowing between closeby objects, as well as darkening in corners, creases, and cracks. The scattering models associated with objects in the scene are ignored.

8.10.2. Direct illumination integrator (`direct`)

Parameter	Type	Description
`shadingSamples`	`integer`	This convenience parameter can be used to set both `emitterSamples` and `bsdfSamples` at the same time.
`emitterSamples`	`integer`	Optional more fine-grained parameter: specifies the number of samples that should be generated using the direct illumination strategies implemented by the scene's emitters (Default: set to the value of `shadingSamples`)
`bsdfSamples`	`integer`	Optional more fine-grained parameter: specifies the number of samples that should be generated using the BSDF sampling strategies implemented by the scene's surfaces (Default: set to the value of `shadingSamples`)
`strictNormals`	`boolean`	Be strict about potential inconsistencies involving shading normals? See page 154 for details. (Default: no, i.e. `false`)
`hideEmitters`	`boolean`	Hide directly visible emitters? See page 149 for details. (Default: no, i.e. `false`)



(a) Only BSDF sampling (b) Only emitter sampling (c) BSDF and emitter sampling

Figure 33: This plugin implements two different strategies for computing the direct illumination on surfaces. Both of them are dynamically combined then obtain a robust rendering algorithm.

This integrator implements a direct illumination technique that makes use of *multiple importance sampling*: for each pixel sample, the integrator generates a user-specifiable number of BSDF and emitter samples and combines them using the power heuristic. Usually, the BSDF sampling technique works very well on glossy objects but does badly everywhere else (Figure 33a), while the opposite is true for the emitter sampling technique (Figure 33b). By combining these approaches, one can obtain a rendering technique that works well in both cases (Figure 33c).

The number of samples spent on either technique is configurable, hence it is also possible to turn this plugin into an emitter sampling-only or BSDF sampling-only integrator.

For best results, combine the direct illumination integrator with the low-discrepancy sample generator (`ldsampler`). Generally, the number of pixel samples of the sample generator can be kept relatively low (e.g. `sampleCount=4`), whereas the `shadingSamples` parameter of this integrator should be increased until the variance in the output renderings is acceptable.

Remarks:

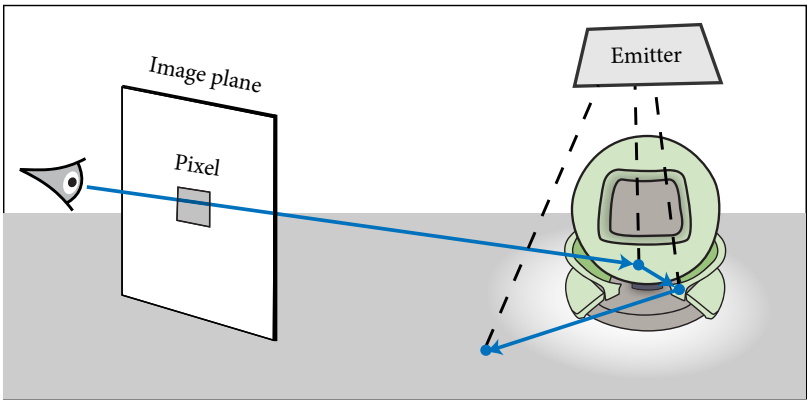
- This integrator does not handle participating media or indirect illumination.

8.10.3. Path tracer (`path`)

Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`rrDepth`	`integer`	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: `5`)
`strictNormals`	`boolean`	Be strict about potential inconsistencies involving shading normals? See the description below for details. (Default: no, i.e. `false`)
`hideEmitters`	`boolean`	Hide directly visible emitters? See page 149 for details. (Default: no, i.e. `false`)

This integrator implements a basic path tracer and is a *good default choice* when there is no strong reason to prefer another method.

To use the path tracer appropriately, it is instructive to know roughly how it works: its main operation is to trace many light paths using *random walks* starting from the sensor. A single random walk is shown below, which entails casting a ray associated with a pixel in the output image and searching for the first visible intersection. A new direction is then chosen at the intersection, and the ray-casting step repeats over and over again (until one of several stopping criteria applies).



At every intersection, the path tracer tries to create a connection to the light source in an attempt to find a *complete* path along which light can flow from the emitter to the sensor. This of course only works when there is no occluding object between the intersection and the emitter.

This directly translates into a category of scenes where a path tracer can be expected to produce reasonable results: this is the case when the emitters are easily "accessible" by the contents of the scene. For instance, an interior scene that is lit by an area light will be considerably harder to render when this area light is inside a glass enclosure (which effectively counts as an occluder).

Like the `direct` plugin, the path tracer internally relies on multiple importance sampling to combine BSDF and emitter samples. The main difference in comparison to the former plugin is that it considers light paths of arbitrary length to compute both direct and indirect illumination.

For good results, combine the path tracer with one of the low-discrepancy sample generators (i.e. `ldsampler`, `halton`, or `sobol`).

Strict normals: Triangle meshes often rely on interpolated shading normals to suppress the inherently faceted appearance of the underlying geometry. These "fake" normals are not without problems, however. They can lead to paradoxical situations where a light ray impinges on an object from a direction that is classified as "outside" according to the shading normal, and "inside" according to the true geometric normal.

The `strictNormals` parameter specifies the intended behavior when such cases arise. The default (`false`, i.e. "carry on") gives precedence to information given by the shading normal and considers such light paths to be valid. This can theoretically cause light "leaks" through boundaries, but it is not much of a problem in practice.

When set to `true`, the path tracer detects inconsistencies and ignores these paths. When objects are poorly tesselated, this latter option may cause them to lose a significant amount of the incident radiation (or, in other words, they will look dark).

The bidirectional integrators in Mitsuba (`bdpt`, `pssmlt`, `mlt` ...) implicitly have `strictNormals` set to `true`. Hence, another use of this parameter is to match renderings created by these methods.

Remarks:

- This integrator does not handle participating media
- This integrator has poor convergence properties when rendering caustics and similar effects. In this case, `bdpt` or one of the photon mappers may be preferable.

8.10.4. Simple volumetric path tracer (`volpath_simple`)

Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`rrDepth`	`integer`	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: `5`)
`strictNormals`	`boolean`	Be strict about potential inconsistencies involving shading normals? See page 154 for details. (Default: no, i.e. `false`)
`hideEmitters`	`boolean`	Hide directly visible emitters? See page 149 for details. (Default: no, i.e. `false`)

This plugin provides a basic volumetric path tracer that can be used to compute approximate solutions of the radiative transfer equation. This particular integrator is named "simple" because it does not make use of multiple importance sampling. This results in a potentially faster execution time. On the other hand, it also means that this plugin will likely not perform well when given a scene that contains highly glossy materials. In this case, please use `volpath` or one of the bidirectional techniques.

This integrator has special support for *index-matched* transmission events (i.e. surface scattering events that do not change the direction of light). As a consequence, participating media enclosed by a stencil shape (see Section 8.1 for details) are rendered considerably more efficiently when this shape has *no*[16] BSDF assigned to it (as compared to, say, a `dielectric` or `roughdielectric` BSDF).

Remarks:

- This integrator performs poorly when rendering participating media that have a different index of refraction compared to the surrounding medium.
- This integrator has difficulties rendering scenes that contain relatively glossy materials (`volpath` is preferable in this case).
- This integrator has poor convergence properties when rendering caustics and similar effects. In this case, `bdpt` or one of the photon mappers may be preferable.

[16] this is what signals to Mitsuba that the boundary is index-matched and does not interact with light in any way. Alternatively, the `mask` and `thindielectric` BSDF can be used to specify index-matched boundaries that involve some amount of interaction.

8.10.5. Extended volumetric path tracer (`volpath`)

Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`rrDepth`	`integer`	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: `5`)
`strictNormals`	`boolean`	Be strict about potential inconsistencies involving shading normals? See page 154 for details. (Default: no, i.e. `false`)
`hideEmitters`	`boolean`	Hide directly visible emitters? See page 149 for details. (Default: no, i.e. `false`)

This plugin provides a volumetric path tracer that can be used to compute approximate solutions of the radiative transfer equation. Its implementation makes use of multiple importance sampling to combine BSDF and phase function sampling with direct illumination sampling strategies. On surfaces, it behaves exactly like the standard path tracer.

This integrator has special support for *index-matched* transmission events (i.e. surface scattering events that do not change the direction of light). As a consequence, participating media enclosed by a stencil shape (see Section 8.1 for details) are rendered considerably more efficiently when this shape has *no*[17] BSDF assigned to it (as compared to, say, a `dielectric` or `roughdielectric` BSDF).

Remarks:

- This integrator will generally perform poorly when rendering participating media that have a different index of refraction compared to the surrounding medium.
- This integrator has poor convergence properties when rendering caustics and similar effects. In this case, `bdpt` or one of the photon mappers may be preferable.

[17] this is what signals to Mitsuba that the boundary is index-matched and does not interact with light in any way. Alternatively, the `mask` and `thindielectric` BSDF can be used to specify index-matched boundaries that involve some amount of interaction.

8.10.6. Bidirectional path tracer (`bdpt`)

Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`lightImage`	`boolean`	Include sampling strategies that connect paths traced from emitters directly to the camera? (i.e. what `ptracer` does) This improves the effectiveness of bidirectional path tracing but severely increases the local and remote communication overhead, since large *light images* must be transferred between threads or over the network. See the text below for a more detailed explanation. (Default: include these strategies, i.e. `true`)
`sampleDirect`	`boolean`	Enable direct sampling strategies? This is a generalization of direct illumination sampling that works with both emitters and sensors. Usually a good idea. (Default: use direct sampling, i.e. `true`)
`rrDepth`	`integer`	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: `5`)

(a) Path tracer, 32 samples/pixel (b) Bidirectional path tracer, 32 samples/pixel

Figure 34: The bidirectional path tracer finds light paths by generating partial paths starting at the emitters and the sensor and connecting them in every possible way. This works particularly well in closed scenes as the one shown above. Here, the unidirectional path tracer has severe difficulties finding some of the indirect illumination paths. Modeled after after a scene by Eric Veach.

This plugin implements a bidirectional path tracer (short: BDPT) with support for multiple importance sampling, as proposed by Veach and Guibas [45].

A bidirectional path tracer computes radiance estimates by starting two separate random walks

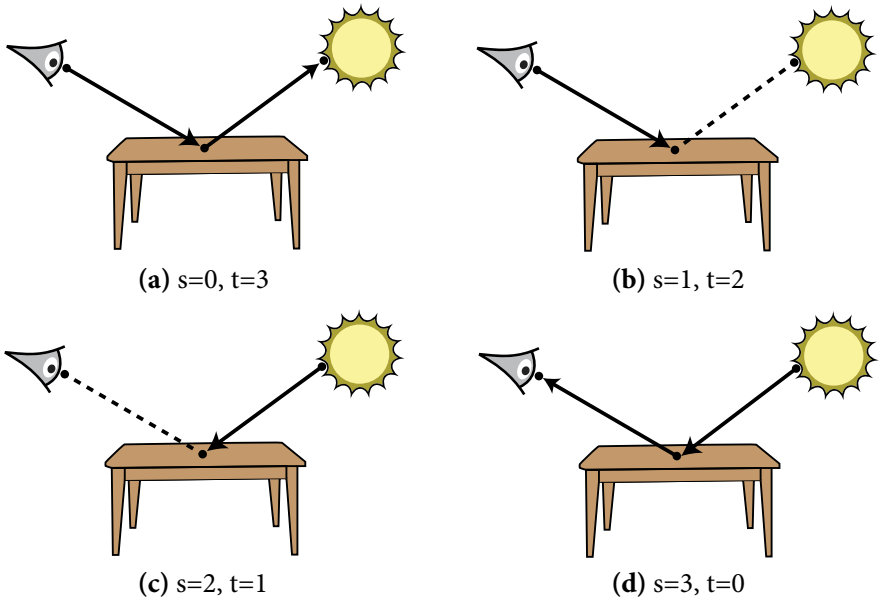


Figure 35: The four different ways in which BDPT can create a direct illumination path (matching the first row on the next page): (**a**) Standard path tracing without direct illumination sampling, (**b**) path tracing with direct illumination sampling, (**c**) Particle tracing with recording of scattering events observed by the sensor, (**d**) Particle tracing with recording of particles that hit the sensor.

from an emitter and a sensor. The resulting *subpaths* are connected at every possible interaction vertex, creating a large number of complete paths of different lengths. These paths are then used to estimate the amount of radiance that is transferred from the emitter to a pixel on the sensor.

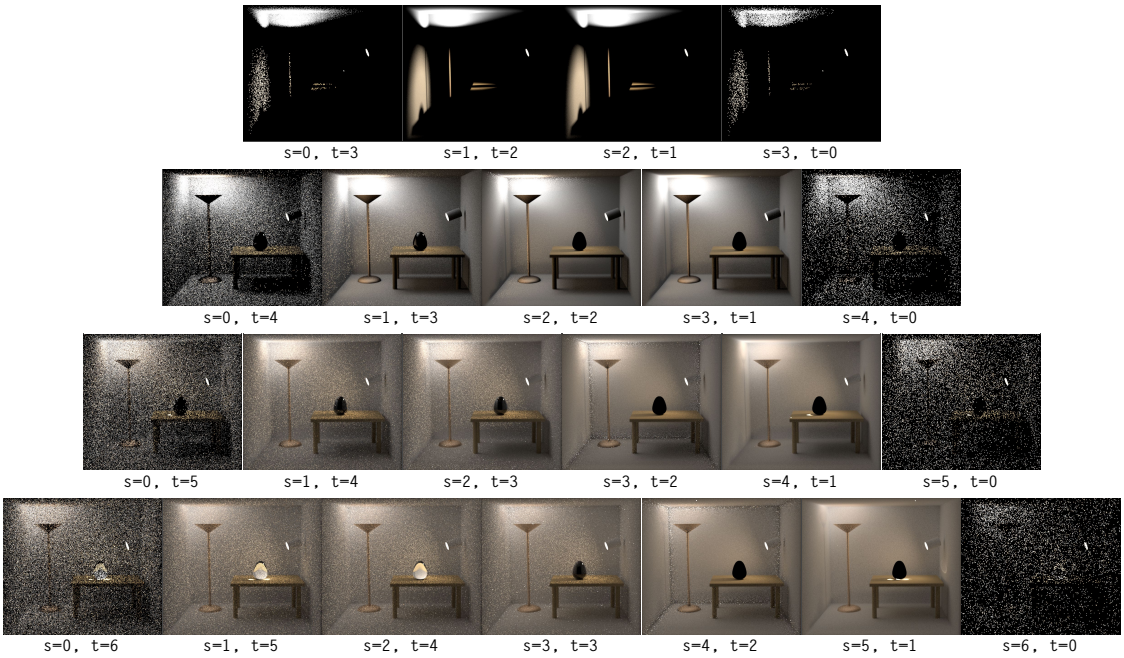
Generally, some of the created paths will be undesirable, since they lead to high-variance radiance estimates. To alleviate this situation, BDPT makes use of *multiple importance sampling* which, roughly speaking, weights paths based on their predicted utility.

The bidirectional path tracer in Mitsuba is a complete implementation of the technique that handles all sampling strategies, including those that involve direct interactions with the sensor. For this purpose, finite-aperture sensors are explicitly represented by surfaces in the scene so that they can be intersected by random walks started at emitters.

Bidirectional path tracing is a relatively "heavy" rendering technique—for the same number of samples per pixel, it is easily 3-4 times slower than regular path tracing. However, it usually makes up for this by producing considerably lower-variance radiance estimates (i.e. the output images have less noise).

The code parallelizes over multiple cores and machines, but with one caveat: some of the BDPT path sampling strategies are incompatble with the usual approach of rendering an image tile by tile, since they can potentially contribute to *any* pixel on the screen. This means that each rendering work unit must be associated with a full-sized image! When network render nodes are involved or the resolution of this *light image* is very high, a bottleneck can arise where more work is spent accumulating or transmitting these images than actual rendering.

There are two possible resorts should this situation arise: the first one is to reduce the number of work units so that there is approximately one unit per core (and hence one image to transmit per core). This can be done by increasing the block size in the GUI preferences or passing the `-b` parameter to the `mitsuba` executable. The second option is to simply disable these sampling strategies at the cost



(a) The individual sampling strategies that comprise BDPT, but *without* multiple importance sampling. s denotes the number of steps taken from the emitters, and t denotes the number of steps from the sensor. Note how almost every strategy has deficiencies of some kind

(b) The same sampling strategies, but now weighted using multiple importance sampling—effectively "turning off" each strategy where it does not perform well. The final result is computed by summing all of these images.

of reducing the effectiveness of bidirectional path tracing (particularly, when rendering caustics). For this, set `lightImage` to `false`. When rendering an image of a reasonable resolution without network nodes, this is not a big concern, hence these strategies are enabled by default.

Remarks:

- This integrator does not work with dipole-style subsurface scattering models.
- This integrator does not yet work with certain non-reciprocal BSDFs (i.e. `bumpmap`, but this will be addressed in the future

8.10.7. Photon map integrator (photonmapper)

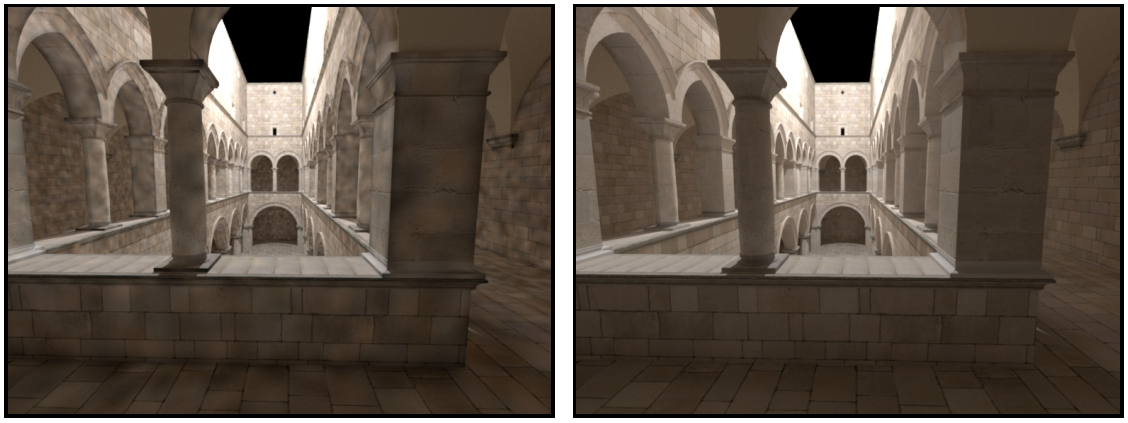
Parameter	Type	Description
directSamples	integer	Number of samples used for the direct illumination component (Default: 16)
glossySamples	integer	Number of samples used for the indirect illumination component of glossy materials (Default: 32)
maxDepth	integer	Specifies the longest path depth in the generated output image (where -1 corresponds to ∞). A value of 1 will only render directly visible light sources. 2 will lead to single-bounce (direct-only) illumination, and so on. (Default: -1)
globalPhotons	integer	Number of photons that will be collected for the global photon map (Default: 250000)
causticPhotons	integer	Number of photons that will be collected for the caustic photon map (Default: 250000)
volumePhotons	integer	Number of photons that will be collected for the volumetric photon map (Default: 250000)
globalLookup↙ Radius	float	Maximum radius of photon lookups in the global photon map (relative to the scene size) (Default: 0.05)
causticLookup↙ Radius	float	Maximum radius of photon lookups in the caustic photon map (relative to the scene size) (Default: 0.0125)
lookupSize	integer	Number of photons that should be fetched in photon map queries (Default: 120)
granularity	integer	Granularity of photon tracing work units for the purpose of parallelization (in # of shot particles) (Default: 0, i.e. decide automatically)
hideEmitters	boolean	Hide directly visible emitters? See page 149 for details. (Default: no, i.e. false)
rrDepth	integer	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: 5)

This plugin implements the two-pass photon mapping algorithm as proposed by Jensen [21]. The implementation partitions the illumination into three different classes (diffuse, caustic, and volumetric), and builds a separate photon map for each class.

Following this, a standard recursive ray tracing pass is started which performs kernel density estimation using these photon maps. Since the photon maps are visualized directly, the result will appear "blotchy" (Figure 36) unless an extremely large number of photons is used. A simple remedy is to combine the photon mapper with an irradiance cache, which performs *final gathering* to remove these artifacts. Due to its caching nature, the rendering process will be faster as well.

```
<integrator type="irrcache">
    <integrator type="photonmapper"/>
</integrator>
```

Listing 35: Instantiation of a photon mapper with irradiance caching



(a) Rendered using plain photon mapping

(b) Rendered using photon mapping together with irradiance caching

Figure 36: Sponza atrium illuminated by a point light and rendered using 5 million photons. Irradiance caching significantly accelerates the rendering time and eliminates the "blotchy" kernel density estimation artifacts. Model courtesy of Marko Dabrovic.

When the scene contains participating media, the Beam Radiance Estimate [20] by Jarosz et al. is used to estimate the illumination due to volumetric scattering.

Remarks:

- Currently, only homogeneous participating media are supported by this implementation

8.10.8. Progressive photon mapping integrator (ppm)

Parameter	Type	Description
maxDepth	integer	Specifies the longest path depth in the generated output image (where -1 corresponds to ∞). A value of 1 will only render directly visible light sources. 2 will lead to single-bounce (direct-only) illumination, and so on. (Default: -1)
photonCount	integer	Number of photons to be shot per iteration (Default: 250000)
initialRadius	float	Initial radius of gather points in world space units. (Default: 0, i.e. decide automatically)
alpha	float	Radius reduction parameter alpha from the paper (Default: 0.7)
granularity	integer	Granularity of photon tracing work units for the purpose of parallelization (in # of shot particles) (Default: 0, i.e. decide automatically)
rrDepth	integer	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: 5)
maxPasses	integer	Maximum number of passes to render (where -1 corresponds to rendering until stopped manually). (Default: -1)

This plugin implements the progressive photon mapping algorithm by Hachisuka et al. [14]. Progressive photon mapping is a variant of photon mapping that alternates between photon shooting and gathering passes that involve a relatively small (e.g. 250K) numbers of photons that are subsequently discarded.

This is done in a way such that the variance and bias of the resulting output vanish as the number of passes tends to infinity. The progressive nature of this method enables renderings with an effectively arbitrary number of photons without exhausting the available system memory.

The desired sample count specified in the sample generator configuration determines how many photon query points are created per pixel. It should not be set too high, since the rendering time is approximately proportional to this number. For good results, use between 2-4 samples along with the `ldsampler`. Once started, the rendering process continues indefinitely until it is manually stopped.

Remarks:

- Due to the data dependencies of this algorithm, the parallelization is limited to to the local machine (i.e. cluster-wide renderings are not implemented)
- This integrator does not handle participating media
- This integrator does not currently work with subsurface scattering models.

8.10.9. Stochastic progressive photon mapping integrator (sppm)

Parameter	Type	Description
maxDepth	integer	Specifies the longest path depth in the generated output image (where -1 corresponds to ∞). A value of 1 will only render directly visible light sources. 2 will lead to single-bounce (direct-only) illumination, and so on. (Default: -1)
photonCount	integer	Number of photons to be shot per iteration (Default: 250000)
initialRadius	float	Initial radius of gather points in world space units. (Default: 0, i.e. decide automatically)
alpha	float	Radius reduction parameter alpha from the paper (Default: 0.7)
granularity	integer	Granularity of photon tracing work units for the purpose of parallelization (in # of shot particles) (Default: 0, i.e. decide automatically)
rrDepth	integer	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: 5)
maxPasses	integer	Maximum number of passes to render (where -1 corresponds to rendering until stopped manually). (Default: -1)

This plugin implements stochastic progressive photon mapping by Hachisuka et al. [13]. This algorithm is an extension of progressive photon mapping (ppm) that improves convergence when rendering scenes involving depth-of-field, motion blur, and glossy reflections.

Note that the implementation of sppm in Mitsuba ignores the sampler configuration—hence, the usual steps of choosing a sample generator and a desired number of samples per pixel are not necessary. As with ppm, once started, the rendering process continues indefinitely until it is manually stopped.

Remarks:

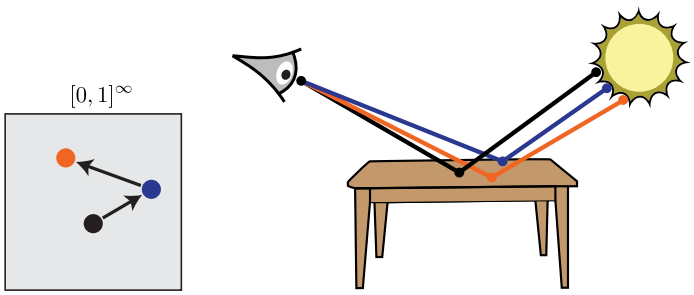
- Due to the data dependencies of this algorithm, the parallelization is limited to to the local machine (i.e. cluster-wide renderings are not implemented)
- This integrator does not handle participating media
- This integrator does not currently work with subsurface scattering models.

8.10.10. Primary Sample Space Metropolis Light Transport (`pssmlt`)

Parameter	Type	Description
`bidirectional`	`boolean`	PSSMLT works in conjunction with another rendering technique that is endowed with Markov Chain-based sample generation. Two choices are available (Default: `true`): • `true`: Operate on top of a fully-fleged bidirectional path tracer with multiple importance sampling. • `false`: Rely on a unidirectional volumetric path tracer (i.e. `volpath`)
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`directSamples`	`integer`	By default, this plugin renders the direct illumination component separately using an optimized direct illumination sampling strategy that uses low-discrepancy number sequences for superior performance (in other words, it is *not* rendered by PSSMLT). This parameter specifies the number of samples allocated to that method. To force PSSMLT to be responsible for the direct illumination component as well, set this parameter to `-1`. (Default: 16)
`rrDepth`	`integer`	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: `5`)
`luminanceSamples`	`integer`	MLT-type algorithms create output images that are only *relative*. The algorithm can e.g. determine that a certain pixel is approximately twice as bright as another one, but the absolute scale is unknown. To recover it, this plugin computes the average luminance arriving at the sensor by generating a number of samples. (Default: `100000` samples)
`twoStage`	`boolean`	Use two-stage MLT? See below for details. (Default: `false`)
`pLarge`	`float`	Rate at which the implementation tries to replace the current path with a completely new one. Usually, there is little need to change this. (Default: 0.3)

Primary Sample Space Metropolis Light Transport (PSSMLT) is a rendering technique developed by Kelemen et al. [26] which is based on Markov Chain Monte Carlo (MCMC) integration. In contrast to simple methods like path tracing that render images by performing a naïve and memoryless random search for light paths, PSSMLT actively searches for *relevant* light paths (as is the case for other MCMC methods). Once such a path is found, the algorithm tries to explore neighboring paths to amortize the cost of the search. This can significantly improve the convergence rate of difficult input. Scenes that were already relatively easy to render usually don't benefit much from PSSMLT, since the MCMC data management causes additional computational overheads.

An interesting aspect of PSSMLT is that it performs this exploration of light paths by perturbing the



(a) Primary sample space view **(b)** Path space view

Figure 37: PSSMLT piggybacks on a rendering method that can turn points in the primary sample space (i.e. "random numbers") into paths. By performing small jumps in primary sample space, it can explore the neighborhood of a path

"random numbers" that were initially used to construct the path. Subsequent regeneration of the path using the perturbed numbers yields a new path in a slightly different configuration, and this process repeats over and over again. The path regeneration step is fairly general and this is what makes the method powerful: in particular, it is possible to use PSSMLT as a layer on top of an existing method to create a new "metropolized" version of the rendering algorithm that is enhanced with a certain degree of adaptiveness as described earlier.

The PSSMLT implementation in Mitsuba can operate on top of either a simple unidirectional volumetric path tracer or a fully-fledged bidirectional path tracer with multiple importance sampling, and this choice is controlled by the `bidirectional` flag. The unidirectional path tracer is generally much faster, but it produces lower-quality samples. Depending on the input, either may be preferable.

Caveats: There are a few general caveats about MLT-type algorithms that are good to know. The first one is that they only render "relative" output images, meaning that there is a missing scale factor that must be applied to obtain proper scene radiance values. The implementation in Mitsuba relies on an additional Monte Carlo estimator to recover this scale factor. By default, it uses 100K samples (controlled by the `luminanceSamples` parameter), which should be adequate for most applications.

The second caveat is that the amount of computational expense associated with a pixel in the output image is roughly proportional to its intensity. This means that when a bright object (e.g. the sun) is visible in a rendering, most resources are committed to rendering the sun disk at the cost of increased variance everywhere else. Since this is usually not desired, the `twoStage` parameter can be used to enable *Two-stage MLT* in this case.

In this mode of operation, the renderer first creates a low-resolution version of the output image to determine the approximate distribution of luminance values. The second stage then performs the actual rendering, while using the previously collected information to ensure that the amount of time spent rendering each pixel is uniform.

The third caveat is that, while PSMLT can work with scenes that are extremely difficult for other methods to handle, it is not particularly efficient when rendering simple things such as direct illumination (which is more easily handled by a brute-force type algorithm). By default, the implementation in Mitsuba therefore delegates this to such a method (with the desired quality being controlled by the `directSamples` parameter). In very rare cases when direct illumination paths are very difficult to find, it is preferable to disable this separation so that PSSMLT is responsible for everything. This can be accomplished by setting `directSamples=-1`.

8.10.11. Path Space Metropolis Light Transport (`mlt`)

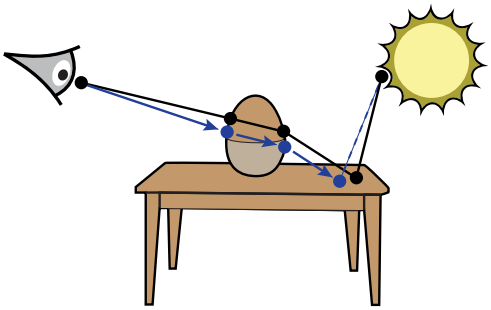
Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`directSamples`	`integer`	By default, the implementation renders direct illumination component separately using the `direct` plugin, which uses low-discrepancy number sequences for superior performance (in other words, it is *not* handled by MLT). This parameter specifies the number of samples allocated to that method. To force MLT to be responsible for the direct illumination component as well, set this to `-1`. (Default: 16)
`luminanceSamples`	`integer`	MLT-type algorithms create output images that are only *relative*. The algorithm can e.g. determine that a certain pixel is approximately twice as bright as another one, but the absolute scale is unknown. To recover it, this plugin computes the average luminance arriving at the sensor by generating a number of samples. (Default: `100000` samples)
`twoStage`	`boolean`	Use two-stage MLT? See `pssmlt` for details. (Default: `false`)
`bidirectionalMutation`, `[lens,multiChain,caustic,manifold]Perturbation`	`boolean`	These parameters can be used to pick the individual mutation and perturbation strategies that will be used to explore path space. By default, the original set by Veach and Guibas is enabled (i.e. everything except the manifold perturbation). It is possible to extend this integrator with additional custom perturbations strategies if needed.
`lambda`	`float`	Jump size of the manifold perturbation (Default: 50)

Metropolis Light Transport (MLT) is a seminal rendering technique proposed by Veach and Guibas [46], which applies the Metropolis-Hastings algorithm to the path-space formulation of light transport. Please refer to the `pssmlt` page for a general description of MLT-type algorithms and a list of caveats that also apply to this plugin.

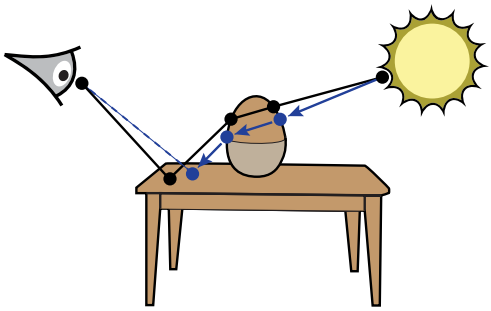
Like `pssmlt`, this integrator explores the space of light paths, searching with preference for those that carry a significant amount of energy from an emitter to the sensor. The main difference is that PSSMLT does this exploration by piggybacking on another rendering technique and "manipulating" the random number stream that drives it, whereas MLT does not use such an indirection: it operates directly on the actual light paths.

This means that the algorithm has access to considerably more information about the problem to be solved, which allows it to perform a directed exploration of certain classes of light paths. The main downside is that the implementation is rather complex, which may make it more susceptible to unforeseen problems. Mitsuba reproduces the full MLT algorithm except for the lens subpath mutation[18]. In addition, the plugin also provides the manifold perturbation proposed by Jakob and Marschner [19].

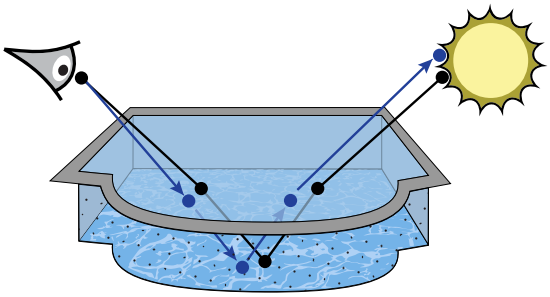
[18] In experiments, it was not found to produce sigificant convergence improvements and was subsequently removed.



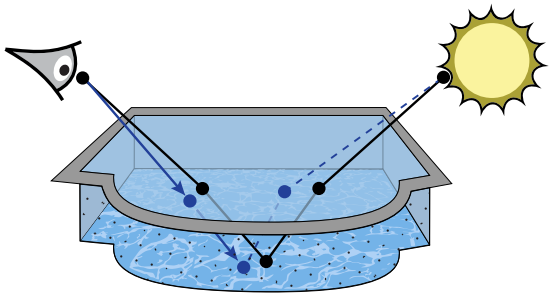
(a) Lens perturbation



(b) Caustic perturbation



(c) Multi-chain perturbation



(d) Manifold perturbation

To explore the space of light paths, MLT iteratively makes changes to a light path, which can either be large-scale *mutations* or small-scale *perturbations*. Roughly speaking, the *bidirectional mutation* is used to jump between different classes of light paths, and each one of the perturbations is responsible for efficiently exploring some of these classes. All mutation and perturbation strategies can be mixed and matched as desired, though for the algorithm to work properly, the bidirectional mutation must be active and perturbations should be selected as required based on the types of light paths that are present in the input scene. The following perturbations are available:

(a) *Lens perturbation*: this perturbation slightly varies the outgoing direction at the camera and propagates the resulting ray until it encounters the first non-specular object. The perturbation then attempts to create a connection to the (unchanged) remainder of the path.

(b) *Caustic perturbation*: essentially a lens perturbation that proceeds in the opposite direction.

(c) *Multi-chain perturbation*: used when there are several chains of specular interactions, as seen in the swimming pool example above. After an initial lens perturbation, a cascade of additional perturbations is required until a connection to the remainder of the path can finally be established. Depending on the path type, the entire path may be changed by this.

(d) *Manifold perturbation*: this perturbation was designed to subsume and extend the previous three approaches. It creates a perturbation at an arbitrary position along the path, proceeding in either direction. Upon encountering a chain of specular interactions, it numerically solves for a connection path (as opposed to the cascading mechanism employed by the multi-chain perturbation).

8.10.12. Energy redistribution path tracing (`erpt`)

Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`numChains`	`float`	On average, how many Markov Chains should be started per pixel? (Default: 1)
`maxChains`	`float`	How many Markov Chains should be started *at most* (per pixel) (Default: `0`, i.e. this feature is not used)
`chainLength`	`integer`	Specifies the number of perturbation steps that are executed per Markov Chain (Default: 1).
`directSamples`	`integer`	By default, the implementation renders direct illumination component separately using the `direct` plugin, which uses low-discrepancy number sequences for superior performance (in other words, it is *not* handled by ERPT). This parameter specifies the number of samples allocated to that method. To force MLT to be responsible for the direct illumination component as well, set this to `-1`. (Default: `16`)
`[lens,multiChain, caustic,manifold]Perturbation`	`boolean`	These parameters can be used to pick the individual perturbation strategies that will be used to explore path space. By default, the original set by Veach and Guibas is enabled (i.e. everything except the manifold perturbation).
`lambda`	`float`	Jump size of the manifold perturbation (Default: `50`)

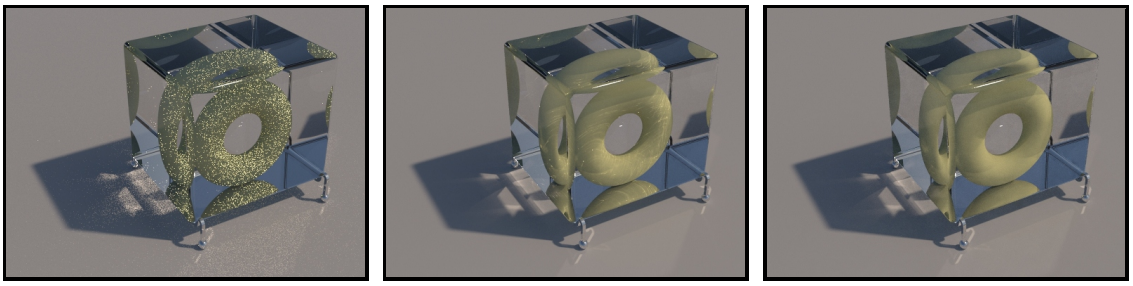
(a) A brass chandelier with 24 glass-enclosed bulbs

(b) Glossy reflective and refractive ableware, lit by the chandelier on the left

Figure 38: An interior scene with complex specular and near-specular light paths, illuminated entirely through caustics. Rendered by this plugin using the manifold perturbation. This scene was designed by Olesya Isaenko.

Energy Redistribution Path Tracing (ERPT) by Cline et al. [5] combines Path Tracing with the perturbation strategies of Metropolis Light Transport.

An initial set of *seed paths* is generated using a standard bidirectional path tracer, and for each one, a MLT-style Markov Chain is subsequently started and executed for some number of steps. This has



(a) Seed paths generated using bidirectional path tracing. Note the high variance of paths that involve reflection of sunlight by the torus.

(b) Result after running the perturbations of Veach and Guibas for 800 steps. Some convergence issues remain.

(c) Result after running the manifold perturbation for the same amount of time

the effect of redistributing the energy of the individual samples over a larger area, hence the name of this method.

This is often a good choice when a (bidirectional) path tracer produces mostly reasonable results except that it finds certain important types of light paths too rarely. ERPT can then explore all of the neighborhing paths as well, to prevent the original sample from showing up as a "bright pixel" in the output image.

Figure 39: Another view, now with exterior lighting.

This plugin shares all the perturbation strategies of the `mlt` plugin, and the same rules for selecting them apply. In contrast to the original paper by Cline et al., the Mitsuba implementation uses a bidirectional (rather than an unidirectional) bidirectional path tracer to create seed paths. Also, since they add bias to the output, this plugin does not use the image post-processing filters proposed by the authors.

The mechanism for selecting Markov Chain seed paths deserves an explanation: when commencing work on a pixel in the output image, the integrator first creates a pool of seed path candidates. The size of this pool is given by the `samplesPerPixel` parameter of the sample generator. This should be large enough so that the integrator has a representative set of light paths to work with.

Subsequently, one or more of these candidates are chosen (determined by `numChains` and `maxChains` parameter). For each one, a Markov Chain is created that has an initial configuration matching the seed path. It is simulated for `chainLength` iterations, and each intermediate state is recorded in the output image.

8.10.13. Adjoint particle tracer (`ptracer`)

Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of `1` will only render directly visible light sources. `2` will lead to single-bounce (direct-only) illumination, and so on. (Default: `-1`)
`rrDepth`	`integer`	Specifies the minimum path depth, after which the implementation will start to use the "russian roulette" path termination criterion. (Default: `5`)
`granularity`	`integer`	Specifies the work unit granularity used to parallize the the particle tracing task. This should be set high enough so that accumulating partially exposed images (and potentially sending them over the network) is not the bottleneck. (Default: 200K particles per work unit, i.e. `200000`)
`bruteForce`	`boolean`	If set to `true`, the integrator does not attempt to create connections to the sensor and purely relies on hitting it via ray tracing. This is mainly intended for debugging purposes. (Default: `false`)

This plugin implements a simple adjoint particle tracer. It does essentially the exact opposite of the simple volumetric path tracer (`volpath_simple`): instead of tracing rays from the sensor and attempting to connect them to the light source, this integrator shoots particles from the light source and attempts to connect them to the sensor.

Usually, this is a relatively useless rendering technique due to its high variance, but there are some cases where it excels. In particular, it does a good job on scenes where most scattering events are directly visible to the camera.

When rendering with a finite-aperture sensor (e.g. `thinlens`) this integrator is able to intersect the actual aperture, which allows it to handle certain caustic paths that would otherwise not be visible.

It also supports a specialized "brute force" mode, where the integrator does not attempt to create connections to the sensor and purely relies on hitting it via ray tracing. This is one of the worst conceivable rendering and not recommended for any applications. It is mainly included for debugging purposes.

The number of traced particles is given by the number of "samples per pixel" of the sample generator times the pixel count of the output image. For instance, 16 samples per pixel on a 512×512 image will cause 4M particles to be generated.

Remarks:

- This integrator does not currently work with subsurface scattering models.

8.10.14. Adaptive integrator (`adaptive`)

Parameter	Type	Description
`maxError`	`float`	Maximum relative error threshold (Default: 0.05)
`pValue`	`float`	Required p-value to accept a sample (Default: 0.05)
`maxSampleFactor`	`integer`	Maximum number of samples to be generated *relative* to the number of configured pixel samples. The adaptive integrator will stop after this many samples, regardless of whether or not the error criterion was satisfied. A negative value will be interpreted as ∞. (Default: 32—for instance, when 64 pixel samples are configured in the `sampler`, this means that the adaptive integrator will give up after 32*64=2048 samples)

This "meta-integrator" repeatedly invokes a provided sub-integrator until the computed radiance values satisfy a specified relative error bound (5% by default) with a certain probability (95% by default). Internally, it uses a Z-test to decide when to stop collecting samples. While repeatedly applying a Z-test in this manner is not good practice in terms of a rigorous statistical analysis, it provides a useful mathematically motivated stopping criterion.

```
<integrator type="adaptive">
    <integrator type="path"/>
</integrator>
```

Listing 36: An example how to make the `path` integrator adaptive

Remarks:

- The adaptive integrator needs a variance estimate to work correctly. Hence, the underlying sample generator should be set to a reasonably large number of pixel samples (e.g. 64 or higher) so that this estimate can be obtained.
- This plugin uses a relatively simplistic error heuristic that does not share information between pixels and only reasons about variance in image space. In the future, it will likely be replaced with something more robust.

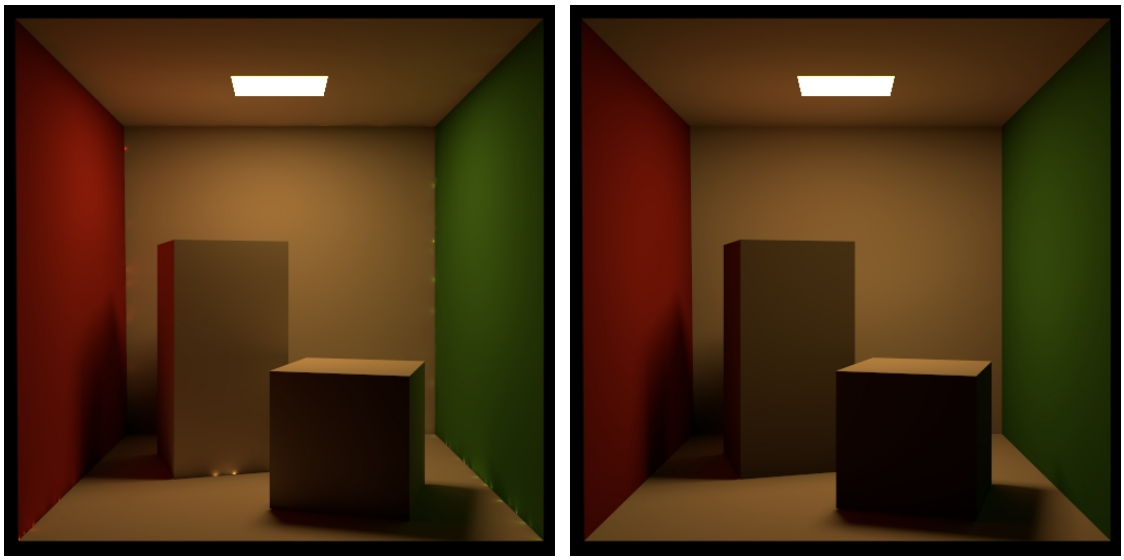
8.10.15. Virtual Point Light integrator (`vpl`)

Parameter	Type	Description
`maxDepth`	`integer`	Specifies the longest path depth in the generated output image (where `-1` corresponds to ∞). A value of 2 will lead to direct-only illumination. (Default: 5)
`shadowMapResolution`	`integer`	Resolution of the shadow maps that are used to compute the point-to-point visibility (Default: 512)
`clamping`	`float`	A relative clamping factor between $[0, 1]$ that is used to control the rendering artifact discussed below. (Default: 0.1)

This integrator implements a hardware-accelerated global illumination rendering technique based on the Instant Radiosity method by Keller [27]. This is the same approach that is also used in Mitsuba's real-time preview; the reason for providing it as a separate integrator plugin is to enable automated (e.g. scripted) usage.

The method roughly works as follows: during a pre-process pass, any present direct and indirect illumination is converted into a set of *virtual point light* sources (VPLs). The scene is then separately rendered many times, each time using a different VPL as a source of illumination. All of the renderings created in this manner are accumulated to create the final output image.

Because the individual rendering steps can be exectuted on a graphics card, it is possible to render many (i.e. 100-1000) VPLs per second. The method is not without problems, however. In particular, it performs poorly when rendering glossy materials, and it produces artifacts in corners and creases . Mitsuba automatically limits the "glossyness" of materials to reduce the effects of the former problem. A `clamping` parameter is provided to control the latter (see the figure below). The number of samples per pixel specified to the sampler is interpreted as the number of VPLs that should be rendered.

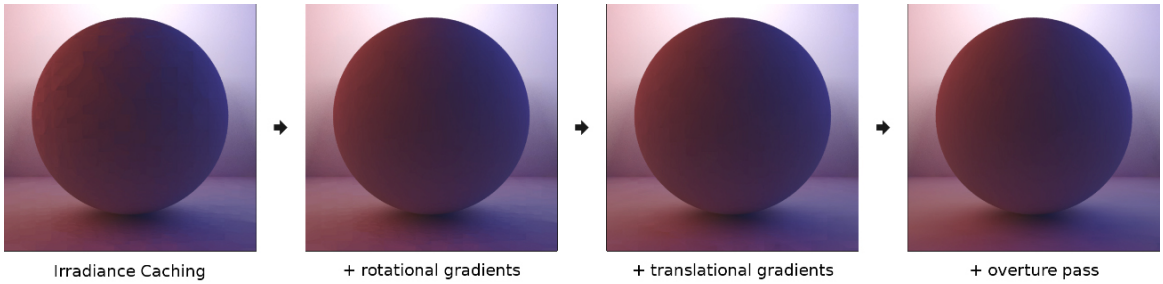


(a) `clamping=0`: With clamping fully disabled, bright blotches appear in corners and creases.

(b) `clamping=0.3`: Higher clamping factors remove these artifacts, but they lead to visible energy loss (the rendering is too dark in certain areas). The default of `0.1` is usually reasonable.

8.10.16. Irradiance caching integrator (`irrcache`)

Parameter	Type	Description
`resolution`	`integer`	Elevational resolution of the stratified final gather hemisphere. The azimuthal resolution is two times this value. (Default: 14, i.e. $2 \cdot 14^2$=392 samples in total)
`quality`	`float`	Quality factor (the κ parameter of Tabellion et al. [43]) (Default: 1.0, which is adequate for most cases)
`gradients`	`boolean`	Use irradiance gradients [49]? (Default: `true`)
`clampNeighbor`	`boolean`	Use neighbor clamping [29]? (Default: `true`)
`clampScreen`	`boolean`	Use a screen-space clamping criterion [43]? (Default: `true`)
`overture`	`boolean`	Do an overture pass before starting the main rendering process? Usually a good idea. (Default: `true`)
`qualityAdjustment`	`float`	When an overture pass is used, Mitsuba subsequently reduces the quality parameter by this amount to interpolate amongst more samples, creating a visually smoother result. (Default: 0.5)
`indirectOnly`	`boolean`	Only show the indirect illumination? This can be useful to check the interpolation quality. (Default: `false`)
`debug`	`boolean`	Visualize the sample placement? (Default: `false`)



(a) Illustration of the effect of the different optimizatations that are provided by this plugin

This "meta-integrator" implements *irradiance caching* by Ward and Heckbert [51]. This method computes and caches irradiance information at a sparse set of scene locations and efficiently determines approximate values at other locations using interpolation.

This plugin only provides the caching and interpolation part—another plugin is still needed to do the actual computation of irradiance values at cache points. This is done using nesting, e.g. as follows:

```
<integrator type="irrcache">
    <integrator type="photonmapper"/>
</integrator>
```

Listing 37: Instantiation of a photon mapper with irradiance caching

When a radiance query involves a non-diffuse material, all computation is forwarded to the sub-

integrator, i.e. `irrcache` is passive. Otherwise, the existing cache points are interpolated to approximate the emitted radiance, or a new cache point is created if the resulting accuracy would be too low. By default, this integrator also performs a distributed overture pass before rendering, which is recommended to avoid artifacts resulting from the addition of samples as rendering proceeds.

Note that wrapping an integrator into `irrcache` adds one extra light bounce. For instance, the method resulting from using `direct` in an irradiance cache renders two-bounce direct illumination.

The generality of this implementation allows it to be used in conjunction with photon mapping (the most likely application) as well as all other sampling-based integrators in Mitsuba. Several optimizations are used to improve the achieved interpolation quality, namely irradiance gradients [49], neighbor clamping [29], a screen-space clamping metric and an improved error function [43].

8.10.17. Multi-channel integrator (`multichannel`)

Parameter	Type	Description
(Nested plugin)	`integrator`	One or more sub-integrators whose output should be rendered into a combined multi-channel image

The multi-channel integrator groups several sub-integrators together and invokes them at the same time for each pixel; the result from each integrator is written into a separate channel of the output image. This could include things like surface normals or the distance from the camera (via the `field` plugin) or ambient occlusion (via the `ao` plugin). In this way, this integrator can be a powerful tool for unusual applications of Mitsuba, e.g. to create reference data for computer vision algorithms. Currently, it only works with a subset of the other plugins—see the red box for details.

Thee `multichannel` plugin also disables certain checks for negative or infinite radiance values during rendering that normally cause warnings to be emitted. This is simply to process extracted fields for which it is fine to take on such values.

The following example contains a typical setup for rendering an 7 channel EXR image: 3 for a path traced image (RGB), 3 for surface normals (encoded as RGB), and 1 channel for the ray distance measured from the camera.

```
<scene>
    <integrator type="multichannel">
        <integrator type="path"/>
        <integrator type="field">
            <string name="field" value="shNormal"/>
        </integrator>
        <integrator type="field">
            <string name="field" value="distance"/>
        </integrator>
    </integrator>

    <sensor type="perspective">
        <sampler type="halton">
            <integer name="sampleCount" value="32"/>
        </sampler>
        <film type="hdrfilm">
            <string name="pixelFormat" value="rgb, rgb, luminance"/>
            <string name="channelNames" value="color, normal, distance"/>
        </film>
    </sensor>
    <!-- **** scene contents **** -->
</scene>
```

Remarks:

- Requires the `hdrfilm` or `tiledhdrfilm`.
- All nested integrators must conform to Mitsuba's basic *SamplingIntegrator* interface. Currently, only a few of them do this, including: `field`, `ao`, `direct`, `path`, `volpath`, `volpath_simple`, and `irrcache`.

8.10.18. Field extraction integrator (`field`)

Parameter	Type	Description
`field`	`string`	Denotes the name of the field that should be extracted. The following choices are possible: • `position`: 3D position in world space • `relPosition`: 3D position in camera space • `distance`: Ray distance to the shading point • `geoNormal`: Geometric surface normal • `shNormal`: Shading surface normal • `uv`: UV coordinate value • `albedo`: Albedo value of the BSDF • `shapeIndex`: Integer index of the high-level shape • `primIndex`: Integer shape primitive index
`undefined`	`spectrum` or `float`	Value that should be returned when there is no intersection (Default: 0)

This integrator extracts a requested field of from the intersection records of shading points and converts the resulting data into color values. It is meant to be used in conjunction with `multichannel` to dump auxiliary information (such as depth or surface normals of surfaces seen by the camera) into extra channels of a rendered image, for instance to create benchmark data for computer vision applications. Please refer to the documentation of `multichannel` for an example.

8.11. Sample generators

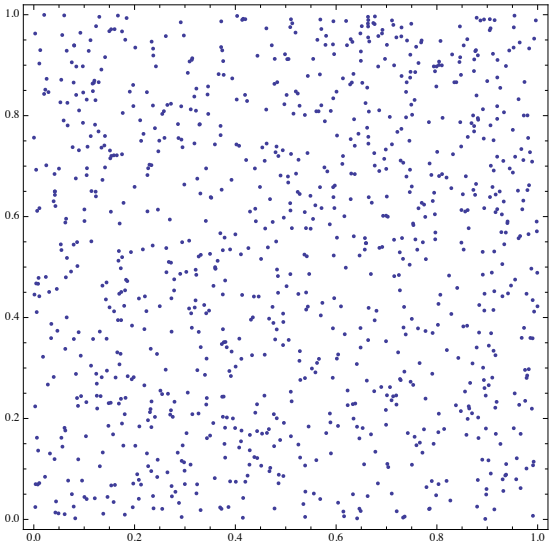
When rendering an image, Mitsuba has to solve a high-dimensional integration problem that involves the geometry, materials, lights, and sensors that make up the scene. Because of the mathematical complexity of these integrals, it is generally impossible to solve them analytically — instead, they are solved *numerically* by evaluating the function to be integrated at a large number of different positions referred to as *samples*. Sample generators are an essential ingredient to this process: they produce points in a (hypothetical) infinite dimensional hypercube $[0,1]^\infty$ that constitute the canonical representation of these samples.

To do its work, a rendering algorithm, or *integrator*, will send many queries to the sample generator. Generally, it will request subsequent 1D or 2D components of this infinite-dimensional "point" and map them into a more convenient space (for instance, positions on surfaces). This allows it to construct light paths to eventually evaluate the flow of light through the scene.

Since the whole process starts with a large number of points in the abstract space $[0,1]^\infty$, it is natural to consider different ways of positioning them. Desirable properties of a sampler are that it "randomly" covers the whole space evenly with samples, but without placing samples too close to each other. This leads to such notions as *stratified sampling* and *low-discrepancy* number sequences. The samplers in this section make different guarantees on the quality of generated samples based on these criteria. To obtain intuition about their behavior, the provided point plots illustrate the resulting sample placement.

8.11.1. Independent sampler (`independent`)

Parameter	Type	Description
`sampleCount`	`integer`	Number of samples per pixel (Default: 4)



(a) A projection of the first 1024 points onto the first two dimensions. Note the sample clumping.

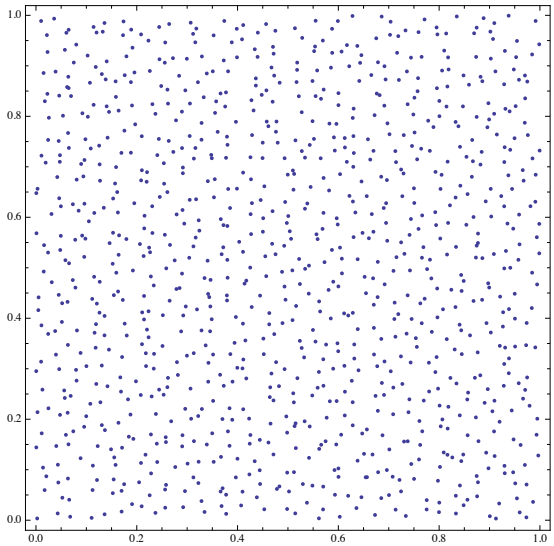
The independent sampler produces a stream of independent and uniformly distributed pseudorandom numbers. Internally, it relies on a fast SIMD version of the Mersenne Twister random number generator [40].

This is the most basic sample generator; because no precautions are taken to avoid sample clumping, images produced using this plugin will usually take longer to converge. In theory, this sampler is initialized using a deterministic procedure, which means that subsequent runs of Mitsuba should create the same image. In practice, when rendering with multiple threads and/or machines, this is not true anymore, since the ordering of samples is influenced by the operating system scheduler.

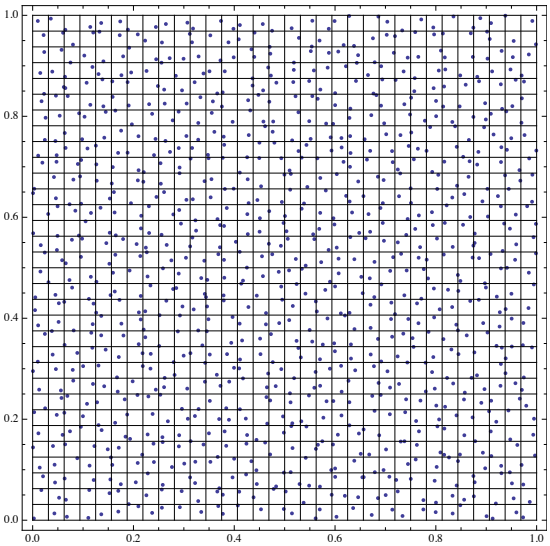
Note that the Metropolis-type integrators implemented in Mitsuba are incompatible with the more sophisticated sample generators shown in this section. They *require* this specific sampler and refuse to work otherwise.

8.11.2. Stratified sampler (`stratified`)

Parameter	Type	Description
`sampleCount`	`integer`	Number of samples per pixel; should be a perfect square (e.g. 1, 4, 9, 16, 25, etc.), or it will be rounded up to the next one (Default: 4)
`dimension`	`integer`	Effective dimension, up to which stratified samples are provided. The number here is to be interpreted as the number of subsequent 1D or 2D sample requests that can be satisfied using "good" samples. Higher high values increase both storage and computational costs. (Default: 4)



(a) A projection of the first 1024 points onto the first two dimensions.



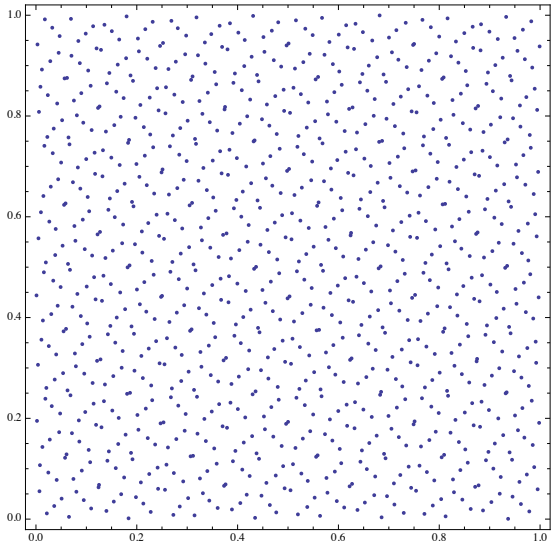
(b) The same samples shown together with the underlying strata for illustrative purposes

The stratified sample generator divides the domain into a discrete number of strata and produces a sample within each one of them. This generally leads to less sample clumping when compared to the independent sampler, as well as better convergence. Due to internal storage costs, stratified samples are only provided up to a certain dimension, after which independent sampling takes over.

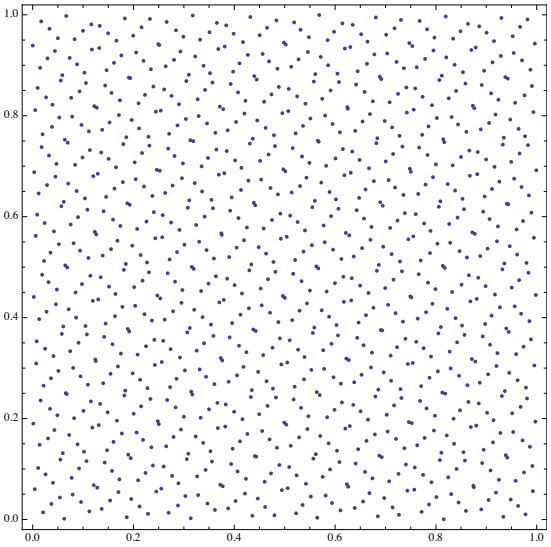
Like the `independent` sampler, multicore and network renderings will generally produce different images in subsequent runs due to the nondeterminism introduced by the operating system scheduler.

8.11.3. Low discrepancy sampler (`ldsampler`)

Parameter	Type	Description
`sampleCount`	`integer`	Number of samples per pixel; should be a power of two (e.g. 1, 2, 4, 8, 16, etc.), or it will be rounded up to the next one (Default: 4)
`dimension`	`integer`	Effective dimension, up to which low discrepancy samples are provided. The number here is to be interpreted as the number of subsequent 1D or 2D sample requests that can be satisfied using "good" samples. Higher high values increase both storage and computational costs. (Default: 4)



(a) A projection of the first 1024 points onto the first two dimensions.



(b) A projection of the first 1024 points onto the 32 and 33th dimension, which look almost identical. However, note that the points have been scrambled to reduce correlations between dimensions.

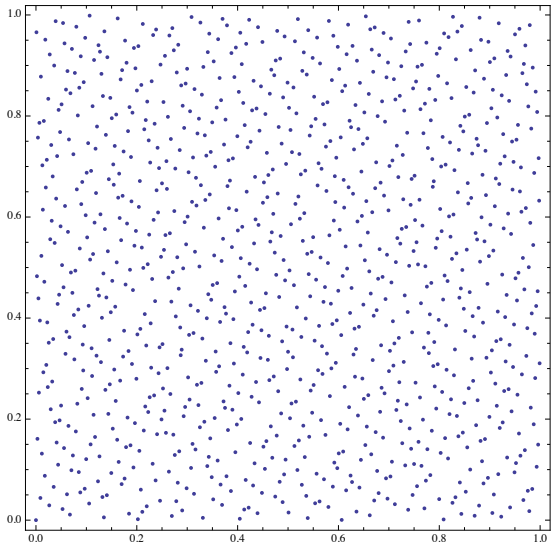
This plugin implements a simple hybrid sampler that combines aspects of a Quasi-Monte Carlo sequence with a pseudorandom number generator based on a technique proposed by Kollig and Keller [28]. It is a good and fast general-purpose sample generator and therefore chosen as the default option in Mitsuba. Some of the QMC samplers in the following pages can generate even better distributed samples, but this comes at a higher cost in terms of performance.

Roughly, the idea of this sampler is that all of the individual 2D sample dimensions are first filled using the same (0, 2)-sequence, which is then randomly scrambled and permuted using numbers generated by a Mersenne Twister pseudorandom number generator [40]. Note that due to internal storage costs, low discrepancy samples are only provided up to a certain dimension, after which independent sampling takes over. The name of this plugin stems from the fact that (0, 2) sequences minimize the so-called *star disrepancy*, which is a quality criterion on their spatial distribution. By now, the name has become slightly misleading since there are other samplers in Mitsuba that just as much try to minimize discrepancy, namely the `sobol` and `halton` plugins.

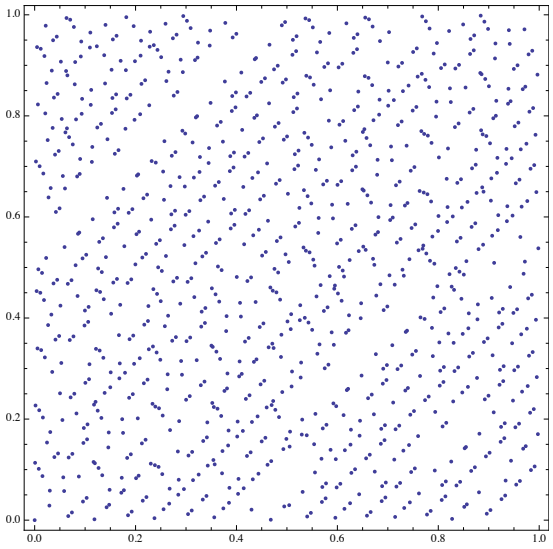
Like the `independent` sampler, multicore and network renderings will generally produce different images in subsequent runs due to the nondeterminism introduced by the operating system scheduler.

8.11.4. Halton QMC sampler (`halton`)

Parameter	Type	Description
`sampleCount`	`integer`	Number of samples per pixel (Default: 4)
`scramble`	`integer`	This plugin can operate in one of three scrambling modes: (i) When set to `0`, the implementation will provide the standard Halton sequence. (ii) When set to `-1`, the implementation will compute a scrambled variant of the Halton sequence based on permutations by Faure [10], which has better equidistribution properties in high dimensions. (iii) When set to a value greater than one, a random permutation is chosen based on this number. This is useful to break up temporally coherent noise when rendering the frames of an animation — in this case, simply set the parameter to the current frame index. Default: `-1`, i.e. use the Faure permutations. Note that permutations rely on a precomputed table that consumes approximately 7 MiB of additional memory at run time.



(a) Projection of the first 1024 points of the Faure-scrambled Halton seq. onto the first two dimensions.

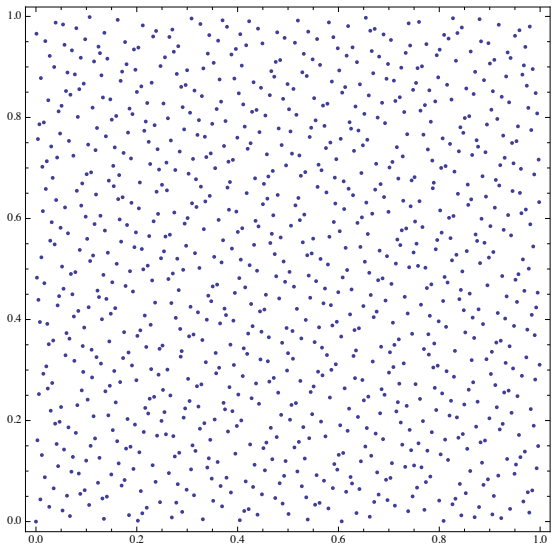


(b) Projection of the first 1024 points of the Faure-scrambled Halton seq. onto the 32th and 33th dim.

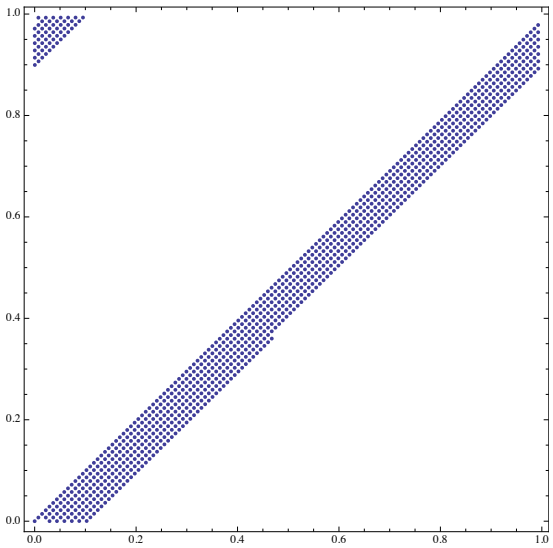
This plugin implements a Quasi-Monte Carlo (QMC) sample generator based on the Halton sequence. QMC number sequences are designed to reduce sample clumping across integration dimensions, which can lead to a higher order of convergence in renderings. Because of the deterministic character of the samples, errors will manifest as grid or moiré patterns rather than random noise, but these diminish as the number of samples is increased.

The Halton sequence in particular provides a very high quality point set that unfortunately becomes increasingly correlated in higher dimensions. To ameliorate this problem, the Halton points

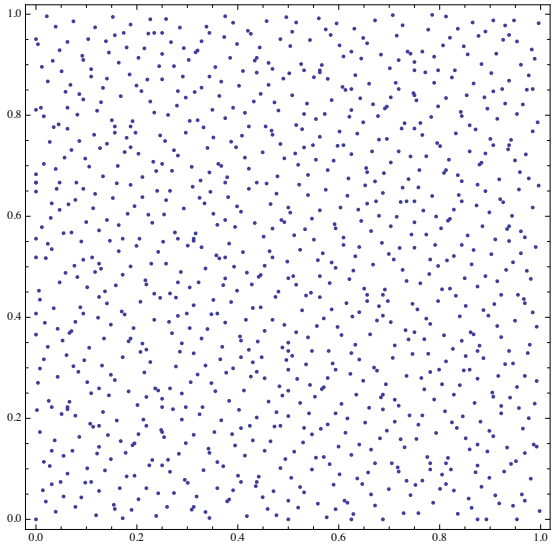
are usually combined with a scrambling permutation, and this is also the default. Because everything that happens inside this sampler is completely deterministic and independent of operating system scheduling behavior, subsequent runs of Mitsuba will always compute the same image, and this even holds when rendering with multiple threads and/or machines.



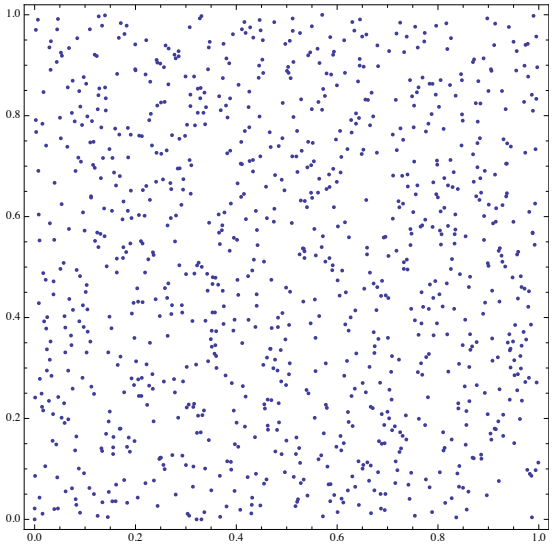
(a) A projection of the first 1024 points of the *original* Halton sequence onto the first two dimensions, obtained by setting `scramble=0`



(b) A projection of the first 1024 points of the *original* Halton sequence onto the 32th and 33th dimensions. Note the strong correlation – a scrambled sequence is usually preferred to avoid this problem.



(a) A projection of the first 1024 points of a randomly scrambled Halton sequence onto the first two dimensions (`scramble=1`).



(b) A projection of the first 1024 points of a randomly scrambled Halton sequence onto the 32th and 33th dimensions.

By default, the implementation provides a scrambled variant of the Halton sequence based on permutations by Faure [10] that has better equidistribution properties in high dimensions, but this can be changed using the `scramble` parameter. Internally, the plugin uses a table of prime numbers

to provide elements of the Halton sequence up to a dimension of 1024. Because of this upper bound, the maximum path depth of the integrator must be limited (e.g. to 100), or rendering might fail with the following error message: *Lookup dimension exceeds the prime number table size! You may have to reduce the 'maxDepth' parameter of your integrator*.

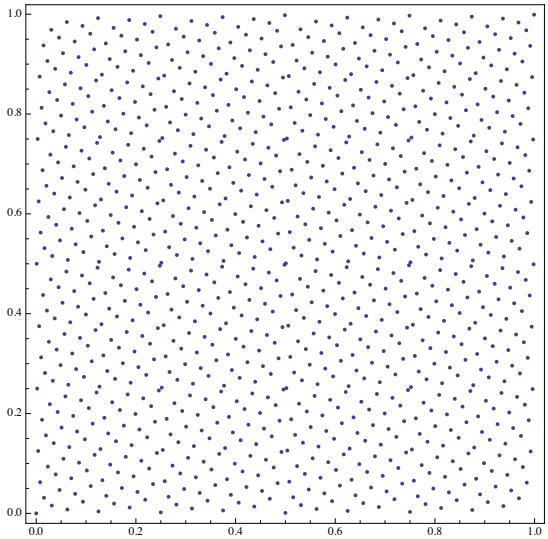
To support bucket-based renderings, the Halton sequence is internally enumerated using a scheme proposed by Grünschloß et al. [12]; the implementation in Mitsuba is based on a Python script by the authors of this paper.

Remarks:

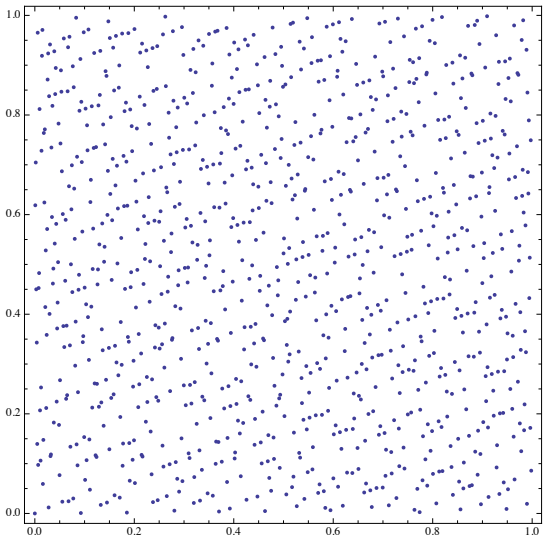
- This sampler is incompatible with Metropolis Light Transport (all variants). It interoperates poorly with Bidirectional Path Tracing and Energy Redistribution Path Tracing, hence these should not be used together. The `sobol` QMC sequence is an alternative for the latter two cases, and `ldsampler` works as well.

8.11.5. Hammersley QMC sampler (`hammersley`)

Parameter	Type	Description
`sampleCount`	`integer`	Number of samples per pixel (Default: 4)
`scramble`	`integer`	This plugin can operate in one of three scrambling modes: (i) When set to `0`, the implementation will provide the standard Hammersley sequence. (ii) When set to `-1`, the implementation will compute a scrambled variant of the Hammersley sequence based on permutations by Faure [10], which has better equidistribution properties in high dimensions. (iii) When set to a value greater than one, a random permutation is chosen based on this number. This is useful to break up temporally coherent noise when rendering the frames of an animation — in this case, simply set the parameter to the current frame index. Default: `-1`, i.e. use the Faure permutations. Note that permutations rely on a precomputed table that consumes approximately 7 MiB of additional memory at run time.



(a) Projection of the first 1024 points of the Faure-scrambled sequence onto the first two dimensions.



(b) Projection of the first 1024 points of the Faure-scrambled sequence onto the 32th and 33th dim.

This plugin implements a Quasi-Monte Carlo (QMC) sample generator based on the Hammersley sequence. QMC number sequences are designed to reduce sample clumping across integration dimensions, which can lead to a higher order of convergence in renderings. Because of the deterministic character of the samples, errors will manifest as grid or moiré patterns rather than random noise, but these diminish as the number of samples is increased.

The Hammerlsey sequence is closely related to the Halton sequence and yields a very high quality point set that is slightly more regular (and has lower discrepancy), especially in the first few dimen-

sions. As is the case with the Halton sequence, the points should be scrambled to reduce patterns that manifest due due to correlations in higher dimensions. Please refer to the halton page for more information on how this works.

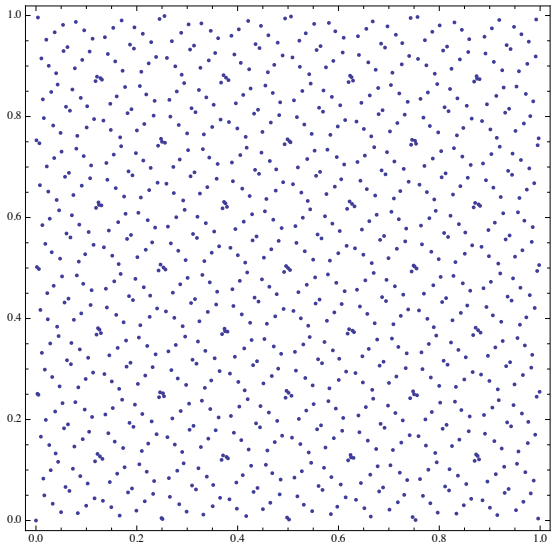
Note that this sampler will cause odd-looking intermediate results when combined with rendering techniques that trace paths starting at light source (e.g. ptracer)—these vanish by the time the rendering process finishes.

Remarks:

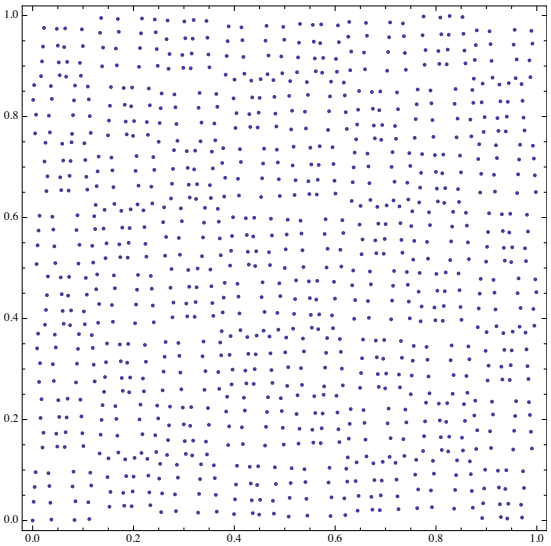
- This sampler is incompatible with Metropolis Light Transport (all variants). It interoperates poorly with Bidirectional Path Tracing and Energy Redistribution Path Tracing, hence these should not be used together. The sobol QMC sequence is an alternative for the latter two cases, and ldsampler works as well.

8.11.6. Sobol QMC sampler (`sobol`)

Parameter	Type	Description
`sampleCount`	`integer`	Number of samples per pixel (Default: 4)
`scramble`	`integer`	This parameter can be used to set a scramble value to break up temporally coherent noise patterns. For stills, this is irrelevant. When rendering an animation, simply set it to the current frame index. (Default: 0)



(a) A projection of the first 1024 points onto the first two dimensions.



(b) A projection of the first 1024 points onto the 32 and 33th dimension.

This plugin implements a Quasi-Monte Carlo (QMC) sample generator based on the Sobol sequence. QMC number sequences are designed to reduce sample clumping across integration dimensions, which can lead to a higher order of convergence in renderings. Because of the deterministic character of the samples, errors will manifest as grid or moiré patterns rather than random noise, but these diminish as the number of samples is increased.

The Sobol sequence in particular provides a relatively good point set that can be computed extremely efficiently. One downside is the susceptibility to pattern artifacts in the generated image. To minimize these artifacts, it is advisable to use a number of samples per pixel that is a power of two.

Because everything that happens inside this sampler is completely deterministic and independent of operating system scheduling behavior, subsequent runs of Mitsuba will always compute the same image, and this even holds when rendering with multiple threads and/or machines.

The plugin relies on a fast implementation of the Sobol sequence by Leonhard Grünschloß using direction numbers provided by Joe and Kuo [24]. These direction numbers are given up to a dimension of 1024. Because of this upper bound, the maximum path depth of the integrator must be limited (e.g. to 100), or rendering might fail with the following error message: *Lookup dimension exceeds the direction number table size! You may have to reduce the 'maxDepth' parameter of your integrator*.

Note that this sampler generates a $(0, 2)$-sequence in the first two dimensions, and therefore the point plot shown in (a) happens to match the corresponding plots of `ldsampler`. In higher dimensions, however, they behave rather differently.

When this sampler is used to perform parallel block-based renderings, the sequence is internally enumerated using a scheme proposed and implemented by Grünschloß et al. [12].

Remarks:

- This sampler is incompatible with Metropolis Light Transport (all variants).

8.12. Films

A film defines how conducted measurements are stored and converted into the final output file that is written to disk at the end of the rendering process. Mitsuba comes with a few films that can write to high and low dynamic range image formats (OpenEXR, JPEG or PNG), as well more scientifically oriented data formats (e.g. MATLAB or Mathematica).

In the XML scene description language, a normal film configuration might look as follows

```
<scene version="0.5.0">
    <!-- ... scene contents ... -->

    <sensor type="... sensor type ...">
        <!-- ... sensor parameters ... -->

        <!-- Write to a high dynamic range EXR image -->
        <film type="hdrfilm">
            <!-- Specify the desired resolution (e.g. full HD) -->
            <integer name="width" value="1920"/>
            <integer name="height" value="1080"/>

            <!-- Use a Gaussian reconstruction filter. For
                 details on these, refer to the next subsection -->
            <rfilter type="gaussian"/>
        </film>
    </sensor>
</scene>
```

The `film` plugin should be instantiated nested inside a `sensor` declaration. Note how the output filename is never specified—it is automatically inferred from the scene filename and can be manually overridden by passing the configuration parameter `-o` to the `mitsuba` executable when rendering from the command line.

8.12.1. High dynamic range film (hdrfilm)

Parameter	Type	Description
width, height	integer	Width and height of the camera sensor in pixels (Default: 768, 576)
fileFormat	string	Denotes the desired output file format. The options are openexr (for ILM's OpenEXR format), rgbe (for Greg Ward's RGBE format), or pfm (for the Portable Float Map format) (Default: openexr)
pixelFormat	string	Specifies the desired pixel format of output images. The options are luminance, luminanceAlpha, rgb, rgba, xyz, xyza, spectrum, and spectrumAlpha. For the spectrum* options, the number of written channels depends on the value assigned to SPECTRUM_SAMPLES during compilation (see Section 4 for details) (Default: rgb)
componentFormat	string	Specifies the desired floating point component format of output images. The options are float16, float32, or uint32. (Default: float16)
cropOffsetX, cropOffsetY, cropWidth, cropHeight	integer	These parameters can optionally be provided to select a subrectangle of the output. In this case, Mitsuba will only render the requested regions. (Default: Unused)
attachLog	boolean	Mitsuba can optionally attach the entire rendering log file as a metadata field so that this information is permanently saved. (Default: true, i.e. attach it)
banner	boolean	Include a small Mitsuba banner in the output image? (Default: true)
highQualityEdges	boolean	If set to true, regions slightly outside of the film plane will also be sampled. This may improve the image quality at the edges, especially when using very large reconstruction filters. In general, this is not needed though. (Default: false, i.e. disabled)
(Nested plugin)	rfilter	Reconstruction filter that should be used by the film. (Default: gaussian, a windowed Gaussian filter)

This is the default film plugin that is used when none is explicitly specified. It stores the captured image as a high dynamic range OpenEXR file and tries to preserve the rendering as much as possible by not performing any kind of post processing, such as gamma correction—the output file will record linear radiance values.

When writing OpenEXR files, the film will either produce a luminance, luminance/alpha, RGB(A), XYZ(A) tristimulus, or spectrum/spectrum-alpha-based bitmap having a float16, float32, or uint32-based internal representation based on the chosen parameters. The default configuration is RGB with a float16 component format, which is appropriate for most purposes. Note that the spectral output options only make sense when using a custom build of Mitsuba that has spectral rendering enabled (this is not the case for the downloadable release builds). For OpenEXR files, Mitsuba

also supports fully general multi-channel output; refer to the `multichannel` plugin for details on how this works.

The plugin can also write RLE-compressed files in the Radiance RGBE format pioneered by Greg Ward (set `fileFormat=rgbe`), as well as the Portable Float Map format (set `fileFormat=pfm`). In the former case, the `componentFormat` and `pixelFormat` parameters are ignored, and the output is "`float8`"-compressed RGB data. PFM output is restricted to `float32`-valued images using the `rgb` or `luminance` pixel formats. Due to the superior accuracy and adoption of OpenEXR, the use of these two alternative formats is discouraged however.

When RGB(A) output is selected, the measured spectral power distributions are converted to linear RGB based on the CIE 1931 XYZ color matching curves and the ITU-R Rec. BT.709-3 primaries with a D65 white point.

```
<film type="hdrfilm">
    <string name="pixelFormat" value="rgba"/>
    <integer name="width" value="1920"/>
    <integer name="height" value="1080"/>
    <boolean name="banner" value="false"/>
</film>
```

Listing 38: Instantiation of a film that writes a full-HD RGBA OpenEXR file without the Mitsuba banner

Render-time annotations:

The `ldrfilm` and `hdrfilm` plugins support a feature referred to as *render-time annotations* to facilitate record keeping. Annotations are used to embed useful information inside a rendered image so that this information is later available to anyone viewing the image. Exemplary uses of this feature might be to store the frame or take number, rendering time, memory usage, camera parameters, or other relevant scene information.

Currently, two different types are supported: a `metadata` annotation creates an entry in the metadata table of the image, which is preferable when the image contents should not be touched. Alternatively, a `label` annotation creates a line of text that is overlaid on top of the image. Note that this is only visible when opening the output file (i.e. the line is not shown in the interactive viewer). The syntax of this looks as follows:

```
<film type="hdrfilm">
    <!-- Create a new metadata entry 'my_tag_name' and set it to the
         value 'my_tag_value' -->
    <string name="metadata['key_name']" value="Hello!"/>

    <!-- Add the label 'Hello' at the image position X=50, Y=80 -->
    <string name="label[50, 80]" value="Hello!"/>
</film>
```

The `value="..."` argument may also include certain keywords that will be evaluated and substituted when the rendered image is written to disk. A list all available keywords is provided in Table 6.

Apart from querying the render time, memory usage, and other scene-related information, it is also possible to 'paste' an existing parameter that was provided to another plugin—for instance,the the camera transform matrix would be obtained as `$sensor['toWorld']`. The name of the active integrator plugin is given by `$integrator['type']`, and so on. All of these can be mixed to build

larger fragments, as following example demonstrates. The result of this annotation is shown in Figure 40.

```
<string name="label[10, 10]" value="Integrator: $integrator['type'],
  $film['width']x$film['height'], $sampler['sampleCount'] spp,
  render time: $scene['renderTime'], memory: $scene['memUsage']"/>
```

Figure 40: A demonstration of the label annotation feature given the example string shown above.

`$scene['renderTime']`	Image render time, use `renderTimePrecise` for more digits.
`$scene['memUsage']`	Mitsuba memory usage[19]. Use `memUsagePrecise` for more digits.
`$scene['coreCount']`	Number of local and remote cores working on the rendering job
`$scene['blockSize']`	Block size used to parallelize up the rendering workload
`$scene['sourceFile']`	Source file name
`$scene['destFile']`	Destination file name
`$integrator['..']`	Copy a named integrator parameter
`$sensor['..']`	Copy a named sensor parameter
`$sampler['..']`	Copy a named sampler parameter
`$film['..']`	Copy a named film parameter

Table 6: A list of all special keywords supported by the annotation feature

8.12.2. Tiled high dynamic range film (tiledhdrfilm)

Parameter	Type	Description
width, height	integer	Width and height of the camera sensor in pixels (Default: 768, 576)
cropOffsetX, cropOffsetY, cropWidth, cropHeight	integer	These parameters can optionally be provided to select a sub-rectangle of the output. In this case, Mitsuba will only render the requested regions. (Default: Unused)
pixelFormat	string	Specifies the desired pixel format for OpenEXR output images. The options are luminance, luminanceAlpha, rgb, rgba, xyz, xyza, spectrum, and spectrumAlpha. In the latter two cases, the number of written channels depends on the value assigned to SPECTRUM_SAMPLES during compilation (see Section 4 for details) (Default: rgb)
componentFormat	string	Specifies the desired floating point component format used for the output. The options are float16, float32, or uint32 (Default: float16)
(Nested plugin)	rfilter	Reconstruction filter that should be used by the film. (Default: gaussian, a windowed Gaussian filter)

This plugin implements a camera film that stores the captured image as a *tiled* high dynamic-range OpenEXR file. It is very similar to hdrfilm, the main difference being that it does not keep the rendered image in memory. Instead, image tiles are directly written to disk as they are being rendered, which enables renderings of extremely large output images that would otherwise not fit into memory (e.g. 100K×100K).

When the image can fit into memory, usage of this plugin is discouraged: due to the extra overhead of tracking image tiles, the rendering process will be slower, and the output files also generally do not compress as well as those produced by hdrfilm.

Based on the provided parameter values, the film will either write a luminance, luminance/alpha, RGB(A), XYZ(A) tristimulus, or spectrum/spectrum-alpha-based bitmap having a float16, float32, or uint32-based internal representation. The default is RGB and float16. Note that the spectral output options only make sense when using a custom compiled Mitsuba distribution that has spectral rendering enabled. This is not the case for the downloadable release builds.

When RGB output is selected, the measured spectral power distributions are converted to linear RGB based on the CIE 1931 XYZ color matching curves and the ITU-R Rec. BT.709 primaries with a D65 white point.

Remarks:

- This film is only meant for command line-based rendering. When used with mtsgui, the preview image will be black.
- This plugin is slower than hdrfilm, and therefore should only be used when the output image is too large to fit into system memory.

8.12.3. Low dynamic range film (`ldrfilm`)

Parameter	Type	Description
`width, height`	`integer`	Camera sensor resolution in pixels (Default: 768, 576)
`fileFormat`	`integer`	The desired output file format: `png` or `jpeg`. (Default: `png`)
`pixelFormat`	`string`	Specifies the pixel format of the generated image. The options are `luminance`, `luminanceAlpha`, `rgb` or `rgba` for PNG output and `rgb` or `luminance` for JPEG output.
`tonemapMethod`	`string`	Method used to tonemap recorded radiance values (i) `gamma`: Exposure and gamma correction (default) (ii) `reinhard`: Apply the the tonemapping technique by Reinhard et al. [39] followd by gamma correction.
`gamma`	`float`	The gamma curve applied to correct the output image, where the special value -1 indicates sRGB. (Default: -1)
`exposure`	`float`	When `gamma` tonemapping is active, this parameter specifies an exposure factor in f-stops that is applied to the image before gamma correction (scaling the radiance values by 2^{exposure}). (Default: 0, i.e. do not change the exposure)
`key`	`float`	When `reinhard` tonemapping is active, this parameter in $(0, 1]$ specifies whether a low-key or high-key image is desired. (Default: 0.18, corresponding to a middle-grey)
`burn`	`float`	When `reinhard` tonemapping is active, this parameter in $[0, 1]$ specifies how much highlights can burn out. (Default: 0, i.e. map all luminance values into the displayable range)
`banner`	`boolean`	Include a banner in the output image? (Default: `true`)
`cropOffsetX`, `cropOffsetY`, `cropWidth`, `cropHeight`	`integer`	These parameters can optionally be provided to select a subrectangle of the output. In this case, Mitsuba will only render the requested regions. (Default: Unused)
`highQualityEdges`	`boolean`	If set to `true`, regions slightly outside of the film plane will also be sampled. This may improve image quality at the edges, but is not needed in general. (Default: `false`)
(Nested plugin)	`rfilter`	Reconstruction filter that should be used by the film. (Default: `gaussian`, a windowed Gaussian filter)

This plugin implements a low dynamic range film that can write out 8-bit PNG and JPEG images in various configurations. It provides basic tonemapping techniques to map recorded radiance values into a reasonable displayable range. An alpha (opacity) channel can be written if desired. By default, the plugin writes gamma-corrected PNG files using the sRGB color space and no alpha channel.

This film is a good choice when low dynamic range output is desired and the rendering setup can be configured to capture the relevant portion of the dynamic range reliably enough so that the original HDR data can safely be discarded. When this is not the case, it may be easier to use `hdrfilm` along with the batch tonemapper (Section 5.3.3).

By default, the plugin assumes that no special tonemapping needs to be done and simply applies an exposure multiplier and sRGB gamma curve to the recorded radiance values before converting them to 8 bit. When the dynamic range varies greatly, it may be preferable to use the photographic tonemapping technique by Reinhard et al. [39], which can be activated by setting `tonemapMethod=reinhard`.

Note that the interactive tonemapper that is available in the graphical user interface `mtsgui` interoperates with this plugin. In particular, when saving the scene (*File→Save*), the currently active tonemapper settings are automatically exported into the updated scene file.

The RGB values exported by this plugin correspond to the ITU-R Rec. BT. 709-3 primaries with a D65 white point. When `gamma` is set to `-1` (the default), the output is in the sRGB color space and will display as intended on compatible devices.

Note that this plugin supports render-time *annotations*, which are described on page 191.

8.12.4. MATLAB / Mathematica / NumPy film (`mfilm`)

Parameter	Type	Description
`width, height`	`integer`	Width and height of the sensor in pixels (Default: 1, 1)
`cropOffsetX, cropOffsetY, cropWidth, cropHeight`	`integer`	These parameters can optionally be provided to select a sub-rectangle of the output. In this case, Mitsuba will only render the requested regions. (Default: Unused)
`fileFormat`	`string`	Specifies the desired output format; must be one of `matlab`, `mathematica`, or `numpy`. (Default: `matlab`)
`digits`	`integer`	Number of significant digits to be written (Default: 4)
`variable`	`string`	Name of the target variable (Default: `"data"`)
`pixelFormat`	`string`	Specifies the desired pixel format of the generated image. The options are `luminance`, `luminanceAlpha`, `rgb`, `rgba`, `spectrum`, and `spectrumAlpha`. In the latter two cases, the number of written channels depends on the value assigned to `SPECTRUM_SAMPLES` during compilation (see Section 4 for details) (Default: `luminance`)
`highQualityEdges`	`boolean`	If set to `true`, regions slightly outside of the film plane will also be sampled. This may improve the image quality at the edges, especially when using very large reconstruction filters. In general (and particularly using the default box filter), this is not needed though. (Default: `false`, i.e. disabled)
(Nested plugin)	`rfilter`	Reconstruction filter that should be used by the film. (Default: `box`, a simple box filter)

(a) Importing and tonemapping an image in Mathematica

This plugin provides a camera film that exports spectrum, RGB, XYZ, or luminance values as a matrix to a MATLAB or Mathematica ASCII file or a NumPy binary file. This is useful when running Mitsuba as simulation step as part of a larger virtual experiment. It can also come in handy when verifying parts of the renderer using an automated test suite.

8.13. Reconstruction filters

Image reconstruction filters are responsible for converting a series of radiance samples generated jointly by the *sampler* and *integrator* into the final output image that will be written to disk at the end of a rendering process. This section gives a brief overview of the reconstruction filters that are available in Mitsuba. There is no universally superior filter, and the final choice depends on a trade-off between sharpness, ringing, and aliasing, and computational efficiency.

Desireable properties of a reconstruction filter are that it sharply captures all of the details that are displayable at the requested image resolution, while avoiding aliasing and ringing. Aliasing is the incorrect leakage of high-frequency into low-frequency detail, and ringing denotes oscillation artifacts near discontinuities, such as a light-shadow transiton.

Box filter (`box`): the fastest, but also about the worst possible reconstruction filter, since it is extremely prone to aliasing. It is included mainly for completeness, though some rare situations may warrant its use.

Tent filter (`tent`): Simple tent, or triangle filter. This reconstruction filter never suffers from ringing and usually causes less aliasing than a naive box filter. When rendering scenes with sharp brightness discontinuities, this may be useful; otherwise, negative-lobed filters will be preferable (e.g. Mitchell-Netravali or Lanczos Sinc)

Gaussian filter (`gaussian`): this is a windowed Gaussian filter with configurable standard deviation. It produces pleasing results and never suffers from ringing, but may occasionally introduce too much blurring. When no reconstruction filter is explicitly requested, this is the default choice in Mitsuba.

Mitchell-Netravali filter (`mitchell`): Separable cubic spline reconstruction filter by Mitchell and Netravali [32] This is often a good compromise between sharpness and ringing.

The plugin has two `float`-valued parameters named `B` and `C` that correspond to the two parameters in the original research paper. By default, these are set to the recommended value of 1/3, but can be tweaked if desired.

Catmull-Rom filter (`catmullrom`): This is a special version of the Mitchell-Netravali filter that has the constants `B` and `C` adjusted to produce higher sharpness at the cost of increased susceptibility to ringing.

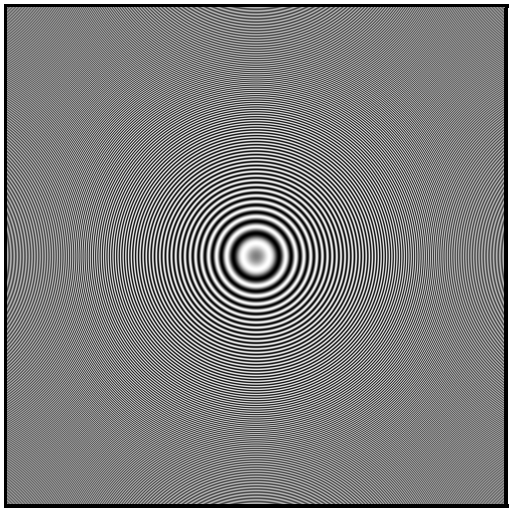
Lanczos Sinc filter (`lanczos`): This is a windowed version of the theoretically optimal low-pass filter. It is generally one of the best available filters in terms of producing sharp high-quality output. Its main disadvantage is that it produces strong ringing around discontinuities, which can become a serious problem when rendering bright objects with sharp edges (for instance, a directly visible light source will have black fringing artifacts around it). This is also the computationally slowest reconstruction filter.

This plugin has an `integer`-valued parameter named `lobes`, that sets the desired number of filter side-lobes. The higher, the closer the filter will approximate an optimal low-pass filter, but this also increases the susceptibility to ringing. Values of 2 or 3 are common (3 is the default).

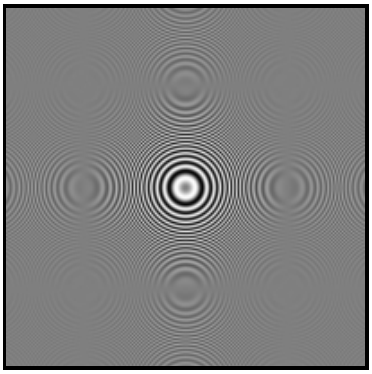
The next section contains a series of comparisons between reconstruction filters. In the first case, a very high-resolution input image (corresponding to a hypothetical radiance field incident at the camera) is reconstructed at low resolutions.

8.13.1. Reconstruction filter comparison 1: frequency attenuation and aliasing

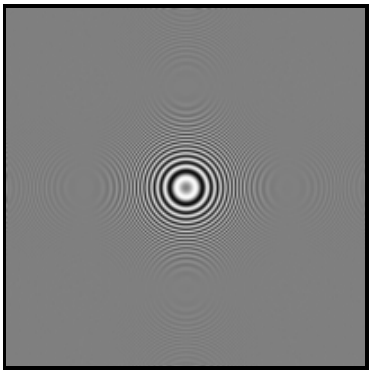
Here, a high frequency function is reconstructed at low resolutions. A good filter (e.g. Lanczos Sinc) will capture all oscillations that are representable at the desired resolution and attenuate the remainder to a uniform gray. The filters are ordered by their approximate level of success at this benchmark.



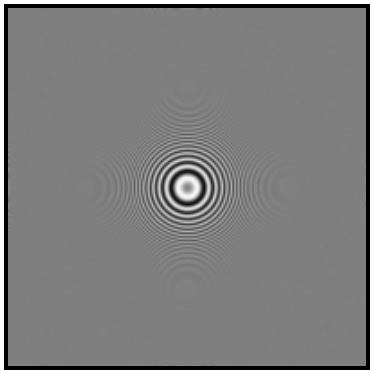
(a) A high resolution input image whose frequency decreases towards the borders. If you are looking at this on a computer, you may have to zoom in.



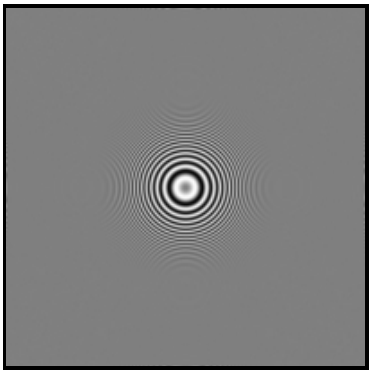
(a) Box filter



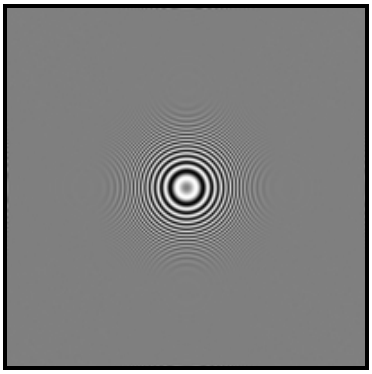
(b) Tent filter



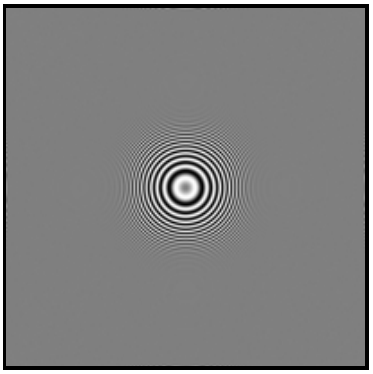
(c) Gaussian filter



(d) Mitchell-Netravali filter



(e) Catmull-Rom filter



(f) Lanczos Sinc filter

8.13.2. Reconstruction filter comparison 2: ringing

This comparison showcases the ringing artifacts that can occur when the rendered image contains extreme and discontinuous brightness transitions. The Mitchell-Netravali, Catmull-Rom, and Lanczos Sinc filters are affected by this problem. Note the black fringing around the light source in the cropped Cornell box renderings below.

(a) Box filter

(b) Tent filter

(c) Gaussian filter

(d) Mitchell-Netravali filter

(e) Catmull-Rom filter

(f) Lanczos Sinc filter

8.13.3. Specifying a reconstruction filter

To specify a reconstruction filter, it must be instantiated inside the sensor's film. Below is an example:

```
<scene version="0.5.0">
    <!-- ... scene contents ... -->

    <sensor type="... sensor type ...">
        <!-- ... sensor parameters ... -->

        <film type="... film type ...">
            <!-- ... film parameters ... -->

            <!-- Instantiate a Lanczos Sinc filter with two lobes -->
            <rfilter type="lanczos">
                <integer name="lobes" value="2"/>
            </rfilter>
        </film>
    </sensor>
</scene>
```

Part II.
Development guide

This chapter and the subsequent ones will provide an overview of the the coding conventions and general architecture of Mitsuba. You should only read them if if you wish to interface with the API in some way (e.g. by developing your own plugins). The coding style section is only relevant if you plan to submit patches that are meant to become part of the main codebase.

9. Code structure

Mitsuba is split into four basic support libraries:

- The core library (`libcore`) implements basic functionality such as cross-platform file and bitmap I/O, data structures, scheduling, as well as logging and plugin management.
- The rendering library (`librender`) contains abstractions needed to load and represent scenes containing light sources, shapes, materials, and participating media.
- The hardware acceleration library (`libhw`) implements a cross-platform display library, an object-oriented OpenGL wrapper, as well as support for rendering interactive previews of scenes.
- Finally, the bidirectional library (`libbidir`) contains a support layer that is used to implement bidirectional rendering algorithms such as Bidirectional Path Tracing and Metropolis Light Transport.

A detailed reference of these APIs is available at `http://www.mitsuba-renderer.org/api`. The next sections present a few basic examples to get familiar with them.

10. Coding style

Indentation: The Mitsuba codebase uses tabs for indentation, which expand to *four* spaces. Please make sure that you configure your editor this way, otherwise the source code layout will look garbled.

Placement of braces: Opening braces should be placed on the same line to make the best use of vertical space, i.e.

```
if (x > y) {
    x = y;
}
```

Placement of spaces: Placement of spaces follows K&R, e.g.

```
if (x == y) {
    ..
} else if (x > y) {
    ..
```

```
} else {
    ..
}
```

rather than things like this

```
if ( x==y ){
}
..
```

Name format: Names are always written in camel-case. Classes and structures start with a capital letter, whereas member functions and attributes start with a lower-case letter. Attributes of classes have the prefix `m_`. Here is an example:

```
class MyClass {
public:
    MyClass(int value) : m_value(value) { }

    inline void setValue(int value) { m_value = value; }
    inline int getValue() const { return m_value; }
private:
    int m_value;
};
```

Enumerations: For clarity, both enumerations types and entries start with a capital **E**, e.g.

```
enum ETristate {
    ENo = 0,
    EYes,
    EMaybe
};
```

Constant methods and parameters: Declare member functions and their parameters as `const` whenever this is possible and properly conveys the semantics.

Inline methods: Always inline trivial pieces of code, such as getters and setters.

Documentation: Headers files should contain Doxygen-compatible documentation. It is also a good idea to add comments to a `.cpp` file to explain subtleties of an implemented algorithm. However, anything pertaining to the API should go into the header file.

Boost: Use the boost libraries whenever this helps to save time or write more compact code.

Classes vs structures: In Mitsuba, classes usually go onto the heap, whereas structures may be allocated both on the stack and the heap.

Classes that derive from `Object` implement a protected virtual deconstructor, which explicitly prevents them from being allocated on the stack. The only way they can be deallocated is using the built-in reference counting. This is done using the `ref<>` template, e.g.

```
if (..) {
    ref<MyClass> instance = new MyClass();
    instance->doSomething()
}   // reference expires, instance will be deallocated
```

Separation of plugins: Mitsuba encourages that plugins are only used via the generic interface they implement. You will find that almost all plugins (e.g. emitters) don't actually provide a header file, hence they can only be accessed using the generic `Emitter` interface they implement. If any kind of special interaction between plugins is needed, this is usually an indication that the generic interface should be extended to accomodate this.

11. Designing a custom integrator plugin

Suppose you want to design a custom integrator to render scenes in Mitsuba. There are two general ways you can do this, and which one you should take mostly depends on the characteristics of your particular integrator.

The framework distinguishes between *sampling-based* integrators and *generic* ones. A sampling-based integrator is able to generate (usually unbiased) estimates of the incident radiance along a specified rays, and this is done a large number of times to render a scene. A generic integrator is more like a black box, where no assumptions are made on how the the image is created. For instance, the VPL renderer uses OpenGL to rasterize the scene using hardware acceleration, which certainly doesn't fit into the sampling-based pattern. For that reason, it must be implemented as a generic integrator.

Generally, if you can package up your code to fit into the `SamplingIntegrator` interface, you should do it, because you'll get parallelization and network rendering essentially for free. This is done by transparently sending instances of your integrator class to all participating cores and assigning small image blocks for each one to work on. Also, sampling-based integrators can be nested within some other integrators, such as an irradiance cache or an adaptive integrator. This cannot be done with generic integrators due to their black-box nature. Note that it is often still possible to parallelize generic integrators, but this involves significantly more work.

In this section, we'll design a rather contrived sampling-based integrator, which renders a monochromatic image of your scene, where the intensity denotes the distance to the camera. But to get a feel for the overall framework, we'll start with an even simpler one, that just renders a solid-color image.

11.1. Basic implementation

In Mitsuba's `src/integrators` directory, create a file named `myIntegrator.cpp`.

```
#include <mitsuba/render/scene.h>

MTS_NAMESPACE_BEGIN

class MyIntegrator : public SamplingIntegrator {
public:
    MTS_DECLARE_CLASS()
};

MTS_IMPLEMENT_CLASS_S(MyIntegrator, false, SamplingIntegrator)
MTS_EXPORT_PLUGIN(MyIntegrator, "A contrived integrator");
MTS_NAMESPACE_END
```

The `scene.h` header file contains all of the dependencies we'll need for now. To avoid conflicts with other libraries, the whole framework is located in a separate namespace named `mitsuba`, and the lines starting with `MTS_NAMESPACE` ensure that our integrator is placed there as well.

The two lines starting with `MTS_DECLARE_CLASS` and `MTS_IMPLEMENT_CLASS` ensure that this class is recognized as a native Mitsuba class. This is necessary to get things like run-time type information, reference counting, and serialization/unserialization support. Let's take a look at the second of these lines, because it contains several important pieces of information:

The suffix `S` in `MTS_IMPLEMENT_CLASS_S` specifies that this is a serializable class, which means that it can be sent over the network or written to disk and later restored. That also implies that certain methods need to be provided by the implementation — we'll add those in a moment.

The three following parameters specify the name of this class (`MyIntegrator`), the fact that it is *not* an abstract class (`false`), and the name of its parent class (`SamplingIntegrator`).

Just below, you can see a line that starts with `MTS_EXPORT_PLUGIN`. As the name suggests, this line is only necessary for plugins, and it ensures that the specified class (`MyIntegrator`) is what you want to be instantiated when somebody loads this plugin. It is also possible to supply a short descriptive string.

Let's add an instance variable and a constructor:

```
public:
    /// Initialize the integrator with the specified properties
    MyIntegrator(const Properties &props) : SamplingIntegrator(props) {
        Spectrum defaultColor;
        defaultColor.fromLinearRGB(0.2f, 0.5f, 0.2f);
        m_color = props.getSpectrum("color", defaultColor);
    }

private:
    Spectrum m_color;
```

This code fragment sets up a default color (a light shade of green), which can be overridden from the scene file. For example, one could instantiate the integrator from an XML document like this

```
<integrator type="myIntegrator">
    <spectrum name="color" value="1.0"/>
</integrator>
```

in which case white would take preference.

Next, we need to add serialization and unserialization support:

```
    /// Unserialize from a binary data stream
    MyIntegrator(Stream *stream, InstanceManager *manager)
     : SamplingIntegrator(stream, manager) {
        m_color = Spectrum(stream);
    }

    /// Serialize to a binary data stream
    void serialize(Stream *stream, InstanceManager *manager) const {
        SamplingIntegrator::serialize(stream, manager);
        m_color.serialize(stream);
    }
```

This makes use of a *stream* abstraction similar in style to Java. A stream can represent various things, such as a file, a console session, or a network communication link. Especially when dealing with multiple machines, it is important to realize that the machines may use different binary representations related to their respective *endianness*. To prevent issues from arising, the `Stream` interface provides many methods for writing and reading small chunks of data (e.g. `writeShort`, `readFloat`, ..), which automatically perform endianness translation. In our case, the `Spectrum` class already provides serialization/unserialization support, so we don't really have to do anything.

Note that it is crucial that your code calls the serialization and unserialization implementations of the superclass, since it will also read/write some information to the stream.

We haven't used the `manager` parameter yet, so here is a quick overview of what it does: if many cases, we don't just want to serialize a single class, but a whole graph of objects. Some may be referenced many times from different places, and potentially there are even cycles. If we just naively called the serialization and unserialization implementation of members recursively within each class, we'd waste much bandwitdth and potentially end up stuck in an infinite recursion.

This is where the instance manager comes in. Every time you want to serialize a heap-allocated object (suppose it is of type `SomeClass`), instead of calling its serialize method, write

```
ref<SomeClass> myObject = ...;
manager->serialize(stream, myObject.get());
```

Later, to unserialize the object from a stream again, write

```
ref<SomeClass> myObject = static_cast<SomeClass *>(manager->getInstance(stream));
```

Behind the scenes, the object manager adds annotations to the data stream, which ensure that you will end up with the exact same reference graph on the remote side, while only one copy of every object is transmitted and no infinite recursion can occur. But we digress – let's go back to our integrator.

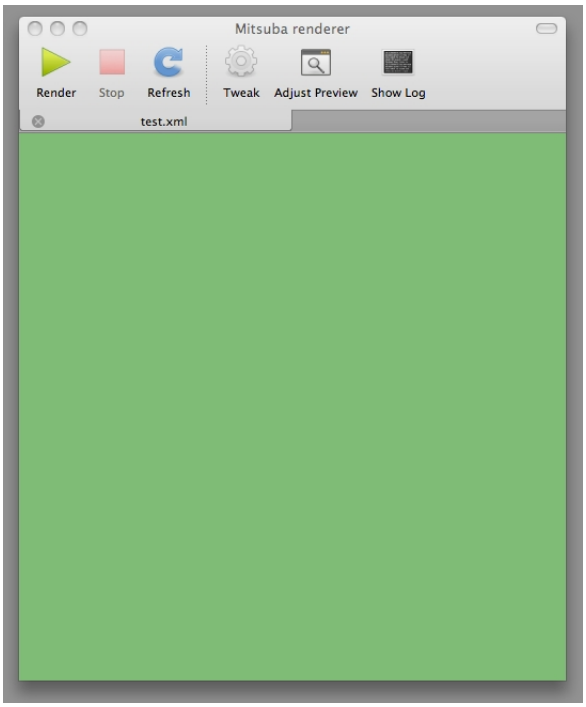
The last thing to add is a function, which returns an estimate for the radiance along a ray differential: here, we simply return the stored color

```
    /// Query for an unbiased estimate of the radiance along <tt>r</tt>
   Spectrum Li(const RayDifferential &r, RadianceQueryRecord &rRec) const {
       return m_color;
   }
```

Let's try building the plugin: edit the `SConscript` file in the `integrator` directory, and add the following line before the last line containing "`Export('plugins')`":

```
plugins += env.SharedLibrary('myIntegrator', ['myIntegrator.cpp'])
```

After calling, `scons`, you should be able to use your new integrator in parallel rendering jobs and you'll get something like this:



That is admittedly not very exciting — so let's do some actual computation.

11.2. Visualizing depth

Add an instance variable `Float m_maxDist;` to the implementation. This will store the maximum distance from the camera to any object, which is needed to map distances into the $[0,1]$ range. Note the upper-case `Float` — this means that either a single- or a double-precision variable is substituted based the compilation flags. This variable constitutes local state, thus it must not be forgotten in the serialization- and unserialization routines: append

```
    m_maxDist = stream->readFloat();
```

and

```
    stream->writeFloat(m_maxDist);
```

to the unserialization constructor and the `serialize` method, respectively.

We'll conservatively bound the maximum distance by measuring the distance to all corners of the bounding box, which encloses the scene. To avoid having to do this every time `Li()` is called, we can override the `preprocess` function:

```
    /// Preprocess function -- called on the initiating machine
    bool preprocess(const Scene *scene, RenderQueue *queue,
            const RenderJob *job, int sceneResID, int cameraResID,
            int samplerResID) {
        SamplingIntegrator::preprocess(scene, queue, job, sceneResID,
            cameraResID, samplerResID);

        const AABB &sceneAABB = scene->getAABB();
    /* Find the camera position at t=0 seconds */
```

```
        Point cameraPosition = scene->getSensor()->getWorldTransform()->eval(0).
    transformAffine(Point(0.0f));
        m_maxDist = - std::numeric_limits<Float>::infinity();

        for (int i=0; i<8; ++i)
            m_maxDist = std::max(m_maxDist,
                (cameraPosition - sceneAABB.getCorner(i)).length());

        return true;
    }
```

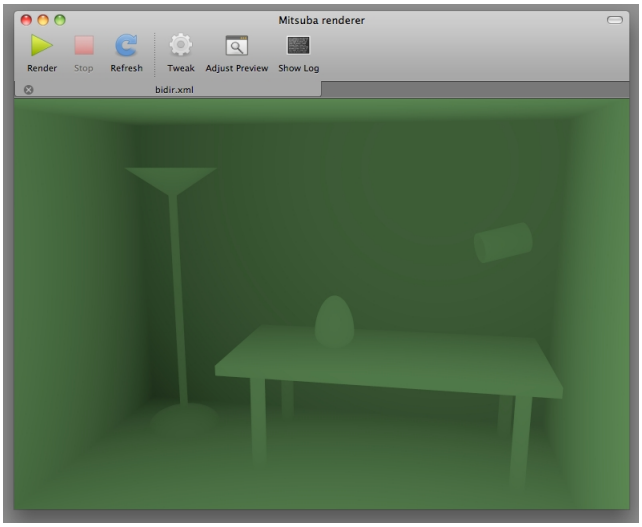
The bottom of this function should be relatively self-explanatory. The numerous arguments at the top are related to the parallelization layer, which will be considered in more detail in the next section. Briefly, the render queue provides synchronization facilities for render jobs (e.g. one can wait for a certain job to terminate). And the integer parameters are global resource identifiers. When a network render job runs, many associated pieces of information (the scene, the camera, etc.) are wrapped into global resource chunks shared amongst all nodes, and these can be referenced using such identifiers.

One important aspect of the `preprocess` function is that it is executed on the initiating node and before any of the parallel rendering begins. This can be used to compute certain things only once. Any information updated here (such as `m_maxDist`) will be forwarded to the other nodes before the rendering begins.

Now, replace the body of the `Li` method with

```
    if (rRec.rayIntersect(r)) {
        Float distance = rRec.its.t;
        return Spectrum(1.0f - distance/m_maxDist) * m_color;
    }
    return Spectrum(0.0f);
```

and the distance renderer is done!



There are a few more noteworthy details: first of all, the "usual" way to intersect a ray against the scene actually works like this:

```
    Intersection its;
```

```
    Ray ray = ...;
    if (scene->rayIntersect(ray, its)) {
        /* Do something with the intersection stored in 'its' */
    }
```

As you can see, we did something slightly different in the distance renderer fragment above (we called `RadianceQueryRecord::rayIntersect()` on the supplied parameter `rRec`), and the reason for this is *nesting*.

11.3. Nesting

The idea of of nesting is that sampling-based rendering techniques can be embedded within each other for added flexibility: for instance, one might concoct a 1-bounce indirect rendering technique complete with irradiance caching and adaptive integration simply by writing the following into a scene XML file:

```
<!-- Adaptively integrate using the nested technique -->
<integrator type="adaptive">
    <!-- Irradiance caching + final gathering with the nested technique -->
    <integrator type="irrcache">
        <!-- Simple direct illumination technique -->
        <integrator type="direct">
    </integrator>
</integrator>
```

To support this kind of complex interaction, some information needs to be passed between the integrators, and the `RadianceQueryRecord` parameter of the function `SamplingIntegrator::Li` is used for this.

This brings us back to the odd way of computing an intersection a moment ago: the reason why we didn't just do this by calling `scene->rayIntersect()` is that our technique might actually be nested within a parent technique, which has already computed this intersection. To avoid wasting resources, the function `rRec.rayIntersect` first determines whether an intersection record has already been provided. If yes, it does nothing. Otherwise, it takes care of computing one.

The radiance query record also lists the particular *types* of radiance requested by the parent integrator – your implementation should respect these as much as possible. Your overall code might for example be structured like this:

```
Spectrum Li(const RayDifferential &r, RadianceQueryRecord &rRec) const {
    Spectrum result;
    if (rRec.type & RadianceQueryRecord::EEmittedRadiance) {
        // Emitted surface radiance contribution was requested
        result += ...;
    }
    if (rRec.type & RadianceQueryRecord::EDirectRadiance) {
        // Direct illumination contribution was requested
        result += ...;
    }
    ...
    return result;
}
```

12. Parallelization layer

Mitsuba is built on top of a flexible parallelization layer, which spreads out various types of computation over local and remote cores. The guiding principle is that if an operation can potentially take longer than a few seconds, it ought to use all the cores it can get.

Here, we will go through a basic example, which will hopefully provide sufficient intuition to realize more complex tasks. To obtain good (i.e. close to linear) speedups, the parallelization layer depends on several key assumptions of the task to be parallelized:

- The task can easily be split up into a discrete number of *work units*, which requires a negligible amount of computation.
- Each work unit is small in footprint so that it can easily be transferred over the network or shared memory.
- A work unit constitutes a significant amount of computation, which by far outweighs the cost of transmitting it to another node.
- The *work result* obtained by processing a work unit is again small in footprint, so that it can easily be transferred back.
- Merging all work results to a solution of the whole problem requires a negligible amount of additional computation.

This essentially corresponds to a parallel version of *Map* (one part of *Map&Reduce*) and is ideally suited for most rendering workloads.

The example we consider here computes a `ROT13` "encryption" of a string, which most certainly violates the "significant amount of computation" assumption. It was chosen due to the inherent parallelism and simplicity of this task. While of course over-engineered to the extreme, the example hopefully communicates how this framework might be used in more complex scenarios.

We will implement this program as a plugin for the utility launcher `mtsutil`, which frees us from having to write lots of code to set up the framework, prepare the scheduler, etc.

We start by creating the utility skeleton file `src/utils/rot13.cpp`:

```
#include <mitsuba/render/util.h>

MTS_NAMESPACE_BEGIN

class ROT13Encoder : public Utility {
public:
    int run(int argc, char **argv) {
        cout << "Hello world!" << endl;
        return 0;
    }

    MTS_DECLARE_UTILITY()
};

MTS_EXPORT_UTILITY(ROT13Encoder, "Perform a ROT13 encryption of a string")
MTS_NAMESPACE_END
```

The file must also be added to the build system: insert the line

```
plugins += env.SharedLibrary('rot13', ['rot13.cpp'])
```

into the `utils/SConscript` file. After compiling using `scons`, the `mtsutil` binary should automatically pick up your new utility plugin:

```
$ mtsutil
..
The following utilities are available:

    addimages             Generate linear combinations of EXR images
    rot13                 Perform a ROT13 encryption of a string
```

It can be executed as follows:

```
$ mtsutil rot13
2010-08-16 18:38:27 INFO  main [src/mitsuba/mtsutil.cpp:276] Mitsuba version 0.1.1,
     Copyright (c) 2010 Wenzel Jakob
2010-08-16 18:38:27 INFO  main [src/mitsuba/mtsutil.cpp:350] Loading utility "
    rot13" ..
Hello world!
```

Our approach for implementing distributed ROT13 will be to treat each character as an indpendent work unit. Since the ordering is lost when sending out work units, we must also include the position of the character in both the work units and the work results.

All of the relevant interfaces are contained in `include/mitsuba/core/sched.h`. For reference, here are the interfaces of `WorkUnit` and `WorkResult`:

```
/**
 * Abstract work unit. Represents a small amount of information
 * that encodes part of a larger processing task.
 */
class MTS_EXPORT_CORE WorkUnit : public Object {
public:
    /// Copy the content of another work unit of the same type
    virtual void set(const WorkUnit *workUnit) = 0;

    /// Fill the work unit with content acquired from a binary data stream
    virtual void load(Stream *stream) = 0;

    /// Serialize a work unit to a binary data stream
    virtual void save(Stream *stream) const = 0;

    /// Return a string representation
    virtual std::string toString() const = 0;

    MTS_DECLARE_CLASS()
protected:
    /// Virtual destructor
    virtual ~WorkUnit() { }
};
/**
 * Abstract work result. Represents the information that encodes
```

```
 * the result of a processed <tt>WorkUnit</tt> instance.
 */
class MTS_EXPORT_CORE WorkResult : public Object {
public:
    /// Fill the work result with content acquired from a binary data stream
    virtual void load(Stream *stream) = 0;

    /// Serialize a work result to a binary data stream
    virtual void save(Stream *stream) const = 0;

    /// Return a string representation
    virtual std::string toString() const = 0;

    MTS_DECLARE_CLASS()
protected:
    /// Virtual destructor
    virtual ~WorkResult() { }
};
```

In our case, the `WorkUnit` implementation then looks like this:

```
class ROT13WorkUnit : public WorkUnit {
public:
    void set(const WorkUnit *workUnit) {
        const ROT13WorkUnit *wu =
            static_cast<const ROT13WorkUnit *>(workUnit);
        m_char = wu->m_char;
        m_pos = wu->m_pos;
    }

    void load(Stream *stream) {
        m_char = stream->readChar();
        m_pos = stream->readInt();
    }

    void save(Stream *stream) const {
        stream->writeChar(m_char);
        stream->writeInt(m_pos);
    }

    std::string toString() const {
        std::ostringstream oss;
        oss << "ROT13WorkUnit[" << endl
            << "  char = '" << m_char << "'," << endl
            << "  pos = " << m_pos << endl
            << "]";
        return oss.str();
    }

    inline char getChar() const { return m_char; }
    inline void setChar(char value) { m_char = value; }
    inline int getPos() const { return m_pos; }
```

```
    inline void setPos(int value) { m_pos = value; }

    MTS_DECLARE_CLASS()
private:
    char m_char;
    int m_pos;
};

MTS_IMPLEMENT_CLASS(ROT13WorkUnit, false, WorkUnit)
```

The `ROT13WorkResult` implementation is not reproduced since it is almost identical (except that it doesn't need the `set` method). The similarity is not true in general: for most algorithms, the work unit and result will look completely different.

Next, we need a class, which does the actual work of turning a work unit into a work result (a subclass of `WorkProcessor`). Again, we need to implement a range of support methods to enable the various ways in which work processor instances will be submitted to remote worker nodes and replicated amongst local threads.

```
class ROT13WorkProcessor : public WorkProcessor {
public:
    /// Construct a new work processor
    ROT13WorkProcessor() : WorkProcessor() { }

    /// Unserialize from a binary data stream (nothing to do in our case)
    ROT13WorkProcessor(Stream *stream, InstanceManager *manager)
        : WorkProcessor(stream, manager) { }

    /// Serialize to a binary data stream (nothing to do in our case)
    void serialize(Stream *stream, InstanceManager *manager) const {
    }

    ref<WorkUnit> createWorkUnit() const {
        return new ROT13WorkUnit();
    }

    ref<WorkResult> createWorkResult() const {
        return new ROT13WorkResult();
    }

    ref<WorkProcessor> clone() const {
        return new ROT13WorkProcessor(); // No state to clone in our case
    }

    /// No internal state, thus no preparation is necessary
    void prepare() { }

    /// Do the actual computation
    void process(const WorkUnit *workUnit, WorkResult *workResult,
                 const bool &stop) {
        const ROT13WorkUnit *wu
            = static_cast<const ROT13WorkUnit *>(workUnit);
        ROT13WorkResult *wr = static_cast<ROT13WorkResult *>(workResult);
```

```
        wr->setPos(wu->getPos());
        wr->setChar((std::toupper(wu->getChar()) - 'A' + 13) % 26 + 'A');
    }
    MTS_DECLARE_CLASS()
};
MTS_IMPLEMENT_CLASS_S(ROT13WorkProcessor, false, WorkProcessor)
```

Since our work processor has no state, most of the implementations are rather trivial. Note the `stop` field in the `process` method. This field is used to abort running jobs at the users requests, hence it is a good idea to periodically check its value during lengthy computations.

Finally, we need a so-called *parallel process* instance, which is responsible for creating work units and stitching work results back into a solution of the whole problem. The `ROT13` implementation might look as follows:

```
class ROT13Process : public ParallelProcess {
public:
    ROT13Process(const std::string &input) : m_input(input), m_pos(0) {
        m_output.resize(m_input.length());
    }

    ref<WorkProcessor> createWorkProcessor() const {
        return new ROT13WorkProcessor();
    }

    std::vector<std::string> getRequiredPlugins() {
        std::vector<std::string> result;
        result.push_back("rot13");
        return result;
    }

    EStatus generateWork(WorkUnit *unit, int worker /* unused */) {
        if (m_pos >= (int) m_input.length())
            return EFailure;
        ROT13WorkUnit *wu = static_cast<ROT13WorkUnit *>(unit);

        wu->setPos(m_pos);
        wu->setChar(m_input[m_pos++]);

        return ESuccess;
    }

    void processResult(const WorkResult *result, bool cancelled) {
        if (cancelled) // indicates a work unit, which was
            return;    // cancelled partly through its execution
        const ROT13WorkResult *wr =
            static_cast<const ROT13WorkResult *>(result);
        m_output[wr->getPos()] = wr->getChar();
    }

    inline const std::string &getOutput() {
        return m_output;
    }
```

```
    MTS_DECLARE_CLASS()
public:
    std::string m_input;
    std::string m_output;
    int m_pos;
};
MTS_IMPLEMENT_CLASS(ROT13Process, false, ParallelProcess)
```

The `generateWork` method produces work units until we have moved past the end of the string, after which it returns the status code `EFailure`. Note the method `getRequiredPlugins()`: this is necessary to use the utility across machines. When communicating with another node, it ensures that the remote side loads the `ROT13*` classes at the right moment.

To actually use the `ROT13` encoder, we must first launch the newly created parallel process from the main utility function (the 'Hello World' code we wrote earlier). We can adapt it as follows:

```
    int run(int argc, char **argv) {
        if (argc < 2) {
            cout << "Syntax: mtsutil rot13 <text>" << endl;
            return -1;
        }

        ref<ROT13Process> proc = new ROT13Process(argv[1]);
        ref<Scheduler> sched = Scheduler::getInstance();

        /* Submit the encryption job to the scheduler */
        sched->schedule(proc);

        /* Wait for its completion */
        sched->wait(proc);

        cout << "Result: " << proc->getOutput() << endl;

        return 0;
    }
```

After compiling everything using `scons`, a simple example involving the utility would be to encode a string (e.g. `SECUREBYDESIGN`), while forwarding all computation to a network machine. (`-p0` disables all local worker threads). Adding a verbose flag (`-v`) shows some additional scheduling information:

```
$ mtsutil -vc feynman -p0 rot13 SECUREBYDESIGN
2010-08-17 01:35:46 INFO  main [src/mitsuba/mtsutil.cpp:201] Mitsuba version 0.1.1,
     Copyright (c) 2010 Wenzel Jakob
2010-08-17 01:35:46 INFO  main [SocketStream] Connecting to "feynman:7554"
2010-08-17 01:35:46 DEBUG main [Thread] Spawning thread "net0_r"
2010-08-17 01:35:46 DEBUG main [RemoteWorker] Connection to "feynman" established
    (2 cores).
2010-08-17 01:35:46 DEBUG main [Scheduler] Starting ..
2010-08-17 01:35:46 DEBUG main [Thread] Spawning thread "net0"
2010-08-17 01:35:46 INFO  main [src/mitsuba/mtsutil.cpp:275] Loading utility "
    rot13" ..
```

```
2010-08-17 01:35:46 DEBUG main [Scheduler] Scheduling process 0: ROT13Process[
    unknown]..
2010-08-17 01:35:46 DEBUG main [Scheduler] Waiting for process 0
2010-08-17 01:35:46 DEBUG net0 [Scheduler] Process 0 has finished generating work
2010-08-17 01:35:46 DEBUG net0_r[Scheduler] Process 0 is complete.
Result: FRPHEROLQRFVTA
2010-08-17 01:35:46 DEBUG main [Scheduler] Pausing ..
2010-08-17 01:35:46 DEBUG net0 [Thread] Thread "net0" has finished
2010-08-17 01:35:46 DEBUG main [Scheduler] Stopping ..
2010-08-17 01:35:46 DEBUG main [RemoteWorker] Shutting down
2010-08-17 01:35:46 DEBUG net0_r[Thread] Thread "net0_r" has finished
```

13. Python integration

A recent feature of Mitsuba is a Python interface to the renderer API. To use this interface, start your Python interpreter and simply enter

```
import mitsuba
```

Mac OS: For this to work on MacOS X, you will first have to run the "*Apple Menu→Command-line access*" menu item from within Mitsuba. If you compiled Mitsuba yourself, then an alternative way of setting the appropriate environment variables without making changes to the system is by sourcing the `setpath.sh` script located in the main Mitsuba directory.

Linux: If you installed one of the official Mitsuba packages for your distribution, then everything should work out of the box. If you compiled Mitsuba yourself, you will need to source the `setpath.sh` script located in the main Mitsuba directory before starting Python.

Windows On Windows it is necessary to explicitly specify the required extension search path within Python before issuing the `import` command, e.g.:

```
import os, sys

# NOTE: remember to specify paths using FORWARD slashes (i.e. '/' instead of
# '\' to avoid pitfalls with string escaping)

# Configure the search path for the Python extension module
sys.path.append('path-to-mitsuba-directory/python/<python version, e.g. 2.7>')

# Ensure that Python will be able to find the Mitsuba core libraries
os.environ['PATH'] = 'path-to-mitsuba-directory' + os.pathsep + os.environ['PATH']

import mitsuba
```

Python API documentation

For an overview of the currently exposed API subset, please refer to the following page: http://www.mitsuba-renderer.org/api/group__libpython.html.

The plugin also exports comprehensive Python-style docstrings, hence the following is an alternative way of getting information on classes, function, or entire namespaces within an interactive Python shell:

```
>>> help(mitsuba.core.Bitmap) # (can be applied to namespaces, classes, functions,
    etc.)

 class Bitmap(Object)
  |  Method resolution order:
  |      Bitmap
  |      Object
  |      Boost.Python.instance
  |      __builtin__.object
```

```
  |
  |  Methods defined here:
  |  __init__(...)
  |      __init__( (object)arg1, (EPixelFormat)arg2, (EComponentFormat)arg3, (
    Vector2i)arg4) -> None :
  |          C++ signature :
  |              void __init__(_object*,mitsuba::Bitmap::EPixelFormat,mitsuba::
    Bitmap::EComponentFormat,mitsuba::TVector2<int>)
  |
  |      __init__( (object)arg1, (EFileFormat)arg2, (Stream)arg3) -> None :
  |          C++ signature :
  |              void __init__(_object*,mitsuba::Bitmap::EFileFormat,mitsuba::
    Stream*)
  |
  |  clear(...)
  |      clear( (Bitmap)arg1) -> None :
  |          C++ signature :
  |              void clear(mitsuba::Bitmap {lvalue})
...
```

The docstrings list the currently exported functionality, as well as C++ and Python signatures, but they don't document what these functions actually do. The web API documentation is the preferred source for this information.

13.1. Basics

Generally, the Python API tries to mimic the C++ API as closely as possible. Where applicable, the Python classes and methods replicate overloaded operators, virtual function calls (which can be overridden in Python), and default arguments. Under rare circumstances, some features are inherently non-portable due to fundamental differences between the two programming languages. In this case, the API documentation will contain further information.

Mitsuba's linear algebra-related classes are usable with essentially the same syntax as their C++ versions — for example, the following snippet creates and rotates a unit vector.

```
import mitsuba
from mitsuba.core import *

# Create a normalized direction vector
myVector = normalize(Vector(1.0, 2.0, 3.0))

# 90 deg. rotation around the Y axis
trafo = Transform.rotate(Vector(0, 1, 0), 90)

# Apply the rotation and display the result
print(trafo * myVector)
```

13.2. Recipes

The following section contains a series of "recipes" on how to do certain things with the help of the Python bindings.

13.2.1. Loading a scene

The following script demonstrates how to use the `FileResolver` and `SceneHandler` classes to load a Mitsuba scene from an XML file:

```
import mitsuba

from mitsuba.core import *
from mitsuba.render import SceneHandler

# Get a reference to the thread's file resolver
fileResolver = Thread.getThread().getFileResolver()

# Register any searchs path needed to load scene resources (optional)
fileResolver.appendPath('<path to scene directory>')

# Optional: supply parameters that can be accessed
# by the scene (e.g. as $myParameter)
paramMap = StringMap()
paramMap['myParameter'] = 'value'

# Load the scene from an XML file
scene = SceneHandler.loadScene(fileResolver.resolve("scene.xml"), paramMap)

# Display a textual summary of the scene's contents
print(scene)
```

13.2.2. Rendering a loaded scene

Once a scene has been loaded, it can be rendered as follows:

```
from mitsuba.core import *
from mitsuba.render import RenderQueue, RenderJob
import multiprocessing

scheduler = Scheduler.getInstance()

# Start up the scheduling system with one worker per local core
for i in range(0, multiprocessing.cpu_count()):
    scheduler.registerWorker(LocalWorker(i, 'wrk%i' % i))
scheduler.start()

# Create a queue for tracking render jobs
queue = RenderQueue()

scene.setDestinationFile('renderedResult')

# Create a render job and insert it into the queue
job = RenderJob('myRenderJob', scene, queue)
job.start()

# Wait for all jobs to finish and release resources
```

```
queue.waitLeft(0)
queue.join()

# Print some statistics about the rendering process
print(Statistics.getInstance().getStats())
```

13.2.3. Rendering over the network

To render over the network, you must first set up one or more machines that run the `mtssrv` server (see Section 5.3.1). A network node can then be registered with the scheduler as follows:

```
# Connect to a socket on a named host or IP address
# 7554 is the default port of 'mtssrv'
stream = SocketStream('128.84.103.222', 7554)

# Create a remote worker instance that communicates over the stream
remoteWorker = RemoteWorker('netWorker', stream)

scheduler = Scheduler.getInstance()
# Register the remote worker (and any other potential workers)
scheduler.registerWorker(remoteWorker)
scheduler.start()
```

13.2.4. Constructing custom scenes from Python

Dynamically constructing Mitsuba scenes entails loading a series of external plugins, instantiating them with custom parameters, and finally assembling them into an object graph. For instance, the following snippet shows how to create a basic perspective sensor with a film that writes PNG images:

```
from mitsuba.core import *
pmgr = PluginManager.getInstance()

# Encodes parameters on how to instantiate the 'perspective' plugin
sensorProps = Properties('perspective')
sensorProps['toWorld'] = Transform.lookAt(
    Point(0, 0, -10),  # Camera origin
    Point(0, 0, 0),    # Camera target
    Vector(0, 1, 0)    # 'up' vector
)
sensorProps['fov'] = 45.0

# Encodes parameters on how to instantiate the 'ldrfilm' plugin
filmProps = Properties('ldrfilm')
filmProps['width'] = 1920
filmProps['height'] = 1080

# Load and instantiate the plugins
sensor = pmgr.createObject(sensorProps)
film = pmgr.createObject(filmProps)

# First configure the film and then add it to the sensor
```

```
film.configure()
sensor.addChild('film', film)

# Now, the sensor can be configured
sensor.configure()
```

The above code fragment uses the plugin manager to construct a `Sensor` instance from an external plugin named `perspective.so/dll/dylib` and adds a child object named `film`, which is a `Film` instance loaded from the plugin `ldrfilm.so/dll/dylib`. Each time after instantiating a plugin, all child objects are added, and finally the plugin's `configure()` method must be called.

Creating scenes in this manner ends up being rather laborious. Since Python comes with a powerful dynamically-typed dictionary primitive, Mitsuba additionally provides a more "pythonic" alternative that makes use of this facility:

```
from mitsuba.core import *

pmgr = PluginManager.getInstance()
sensor = pmgr.create({
    'type' : 'perspective',
    'toWorld' : Transform.lookAt(
        Point(0, 0, -10),
        Point(0, 0, 0),
        Vector(0, 1, 0)
    ),
    'film' : {
        'type' : 'ldrfilm',
        'width' : 1920,
        'height' : 1080
    }
})
```

This code does exactly the same as the previous snippet. By the time `PluginManager.create` returns, the object hierarchy has already been assembled, and the `configure()` method of every object has been called.

Finally, here is an full example that creates a basic scene which can be rendered. It describes a sphere lit by a point light, rendered using the direct illumination integrator.

```
from mitsuba.core import *
from mitsuba.render import Scene

scene = Scene()

# Create a sensor, film & sample generator
scene.addChild(pmgr.create({
    'type' : 'perspective',
    'toWorld' : Transform.lookAt(
        Point(0, 0, -10),
        Point(0, 0, 0),
        Vector(0, 1, 0)
    ),
    'film' : {
        'type' : 'ldrfilm',
```

```
        'width' : 1920,
        'height' : 1080
    },
    'sampler' : {
        'type' : 'ldsampler',
        'sampleCount' : 2
    }
}))

# Set the integrator
scene.addChild(pmgr.create({
    'type' : 'direct'
}))

# Add a light source
scene.addChild(pmgr.create({
    'type' : 'point',
    'position' : Point(5, 0, -10),
    'intensity' : Spectrum(100)
}))

# Add a shape
scene.addChild(pmgr.create({
    'type' : 'sphere',
    'center' : Point(0, 0, 0),
    'radius' : 1.0,
    'bsdf' : {
        'type' : 'diffuse',
        'reflectance' : Spectrum(0.4)
    }
}))

scene.configure()
```

13.2.5. Taking control of the logging system

Many operations in Mitsuba will print one or more log messages during their execution. By default, they will be printed to the console, which may be undesirable. Similar to the C++ side, it is possible to define custom `Formatter` and `Appender` classes to interpret and direct the flow of these messages. This is also useful to keep track of the progress of rendering jobs.

Roughly, a `Formatter` turns detailed information about a logging event into a human-readable string, and a `Appender` routes it to some destination (e.g. by appending it to a file or a log viewer in a graphical user interface). Here is an example of how to activate such extensions:

```
import mitsuba
from mitsuba.core import *

class MyFormatter(Formatter):
    def format(self, logLevel, sourceClass, sourceThread, message, filename, line):

        return '%s (log level: %s, thread: %s, class %s, file %s, line %i)' % \
```

```
                (message, str(logLevel), sourceThread.getName(), sourceClass,
                 filename, line)

class MyAppender(Appender):
    def append(self, logLevel, message):
        print(message)

    def logProgress(self, progress, name, formatted, eta):
        print('Progress message: ' + formatted)

# Get the logger associated with the current thread
logger = Thread.getThread().getLogger()
logger.setFormatter(MyFormatter())
logger.clearAppenders()
logger.addAppender(MyAppender())
logger.setLogLevel(EDebug)

Log(EInfo, 'Test message')
```

13.2.6. Rendering a turntable animation with motion blur

Rendering a turntable animation is a fairly common task that is conveniently accomplished via the Python interface. In a turntable video, the camera rotates around a completely static object or scene. The following snippet does this for the material test ball scene downloadable on the main website, complete with motion blur. It assumes that the scene and scheduler have been set up approriately using one of the previous snippets.

```
sensor = scene.getSensor()
sensor.setShutterOpen(0)
sensor.setShutterOpenTime(1)

stepSize = 5
for i in range(0,360 / stepSize):
    rotationCur  = Transform.rotate(Vector(0, 0, 1), i*stepSize);
    rotationNext = Transform.rotate(Vector(0, 0, 1), (i+1)*stepSize);

    trafoCur  = Transform.lookAt(rotationCur  * Point(0,-6,4),
        Point(0, 0, .5), rotationCur  * Vector(0, 1, 0))
    trafoNext = Transform.lookAt(rotationNext * Point(0,-6,4),
        Point(0, 0, .5), rotationNext * Vector(0, 1, 0))

    atrafo = AnimatedTransform()
    atrafo.appendTransform(0, trafoCur)
    atrafo.appendTransform(1, trafoNext)
    atrafo.sortAndSimplify()
    sensor.setWorldTransform(atrafo)

    scene.setDestinationFile('frame_%03i.png' % i)
    job = RenderJob('job_%i' % i, scene, queue)
    job.start()
```

```
    queue.waitLeft(0)
    queue.join()
```

A useful property of this approach is that scene loading and initialization must only take place once. Performance-wise, this compares favourably with running many separate rendering jobs, e.g. using the `mitsuba` command-line executable.

13.2.7. Simultaneously rendering multiple versions of a scene

Sometimes it is useful to be able to submit multiple scenes to the rendering scheduler at the same time, e.g. when rendering on a big cluster, where one image is not enough to keep all cores on all machines busy. This is is quite easy to do by simply launching multiple `RenderJob` instances before issuing the `queue.waitLeft` call.

However, things go wrong when rendering multiple versions of the *same* scene simultaneously (for instance with a slightly perturbed camera position). The reason for this is that a single `Scene` instance can only be associated with one `RenderJob` at a time. A simple workaround for this is to create a shallow copy that references the original scene, as illustrated in the following snippet:

```
# <Construct scene> in some way
scene.initialize()
sceneResID = scheduler.registerResource(scene)

for i in range(number_of_renderings):
    destination = 'result_%03i' % i

    # Create a shallow copy of the scene so that the queue can tell apart the two
    # rendering processes. This takes almost no extra memory
    newScene = Scene(scene)

    pmgr = PluginManager.getInstance()
    newSensor = pmgr.createObject(scene.getSensor().getProperties())

    # <change the position of 'newSensor' here>

    newFilm = pmgr.createObject(scene.getFilm().getProperties())
    newFilm.configure()
    newSensor.addChild(newFilm)
    newSensor.configure()
    newScene.addSensor(newSensor)
    newScene.setSensor(newSensor)
    newScene.setSampler(scene.getSampler())
    newScene.setDestinationFile(destination)

    # Create a render job and insert it into the queue. Note how the resource
    # ID of the original scene is provided to avoid sending the full scene
    # contents over the network multiple times.
    job = RenderJob('myRenderJob' + str(i), newScene, queue, sceneResID)
    job.start()

# Wait for all jobs to finish and release resources
queue.waitLeft(0)
```

```
queue.join()
```

13.2.8. Creating triangle-based shapes

It is possible to create new triangle-based shapes directly in Python, though doing so is discouraged: because Python is an interpreted programming language, the construction of large meshes will run very slowly. The builtin shapes and shape loaders are to be preferred when this is an option. That said, the following snippet shows how to create `TriMesh` objects from within Python:

```
# Create a new mesh with 1 triangle, 3 vertices,
# and allocate buffers for normals and texture coordinates
mesh = TriMesh('Name of this mesh', 1, 3, True, True)

v = mesh.getVertexPositions()
v[0] = Point3(0, 0, 0)
v[1] = Point3(1, 0, 0)
v[2] = Point3(0, 1, 0)

n = mesh.getVertexNormals()
n[0] = Normal(0, 0, 1)
n[1] = Normal(0, 0, 1)
n[2] = Normal(0, 0, 1)

t = mesh.getTriangles() # Indexed triangle list: tri 1 references vertices 0,1,2
t[0] = 0
t[1] = 1
t[2] = 2

uv = mesh.getTexcoords()
uv[0] = Point2(0, 0)
uv[1] = Point2(1, 0)
uv[2] = Point2(0, 1)

mesh.configure()

# Add to a scene (assumes 'scene' is available)
sensor.addChild(mesh)
```

13.2.9. Calling Mitsuba functions from a multithread Python program

Mitsuba assumes that threads accessing Mitsuba-internal data structures were created by (or at least registered with) Mitsuba. By default, the main thread and subclasses of `mitsuba.core.Thread` satisfy this criterion. But when a Mitsuba function is called from an event dispatch thread of a multi-threaded Python application that is not known to Mitsuba, an exception or crash will usually result.

To avoid this, get a reference to the main thread right after loading the Mitsuba plugin and save some related state (the attached `FileResolver` and `Logger` instances).

```
mainThread = Thread.getThead()
saved_fresolver = mainThread.getFileResolver()
saved_logger = mainThread.getLogger()
```

Later when accessed from an unregister thread, execute the following:

```
# This rendering thread was not created by Mitsuba -- register it
newThread = Thread.registerUnmanagedThread('render')
newThread.setFileResolver(saved_fresolver)
newThread.setLogger(saved_logger)
```

It is fine to execute this several times (`registerUnmanagedThread` just returns a reference to the associated `Thread` instance if it was already registered).

13.2.10. Mitsuba interaction with PyQt/PySide (simple version)

The following listing contains a complete program that renders a sphere and efficiently displays it in a PyQt window (to make this work in PySide, change all occurrences of `PyQt4` to `PySide` in the import declarations and rename the function call to `getNativeBuffer()` to `toByteArray()`, which is a tiny bit less efficient).

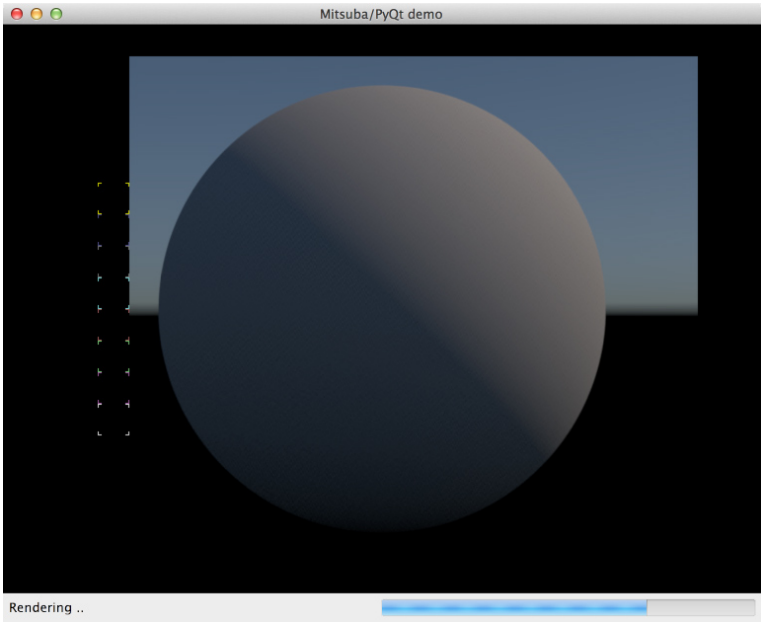
```
import mitsuba, multiprocessing, sys

from mitsuba.core import Scheduler, PluginManager, \
    LocalWorker, Properties, Bitmap, Point2i, FileStream

from mitsuba.render import RenderQueue, RenderJob, Scene

from PyQt4.QtCore import QPoint
from PyQt4.QtGui import QApplication, QMainWindow, QPainter, QImage

class MitsubaView(QMainWindow):
    def __init__(self):
        super(MitsubaView, self).__init__()
        self.setWindowTitle('Mitsuba/PyQt demo')
        self.initializeMitsuba()
        self.image = self.render(self.createScene())
        self.resize(self.image.width(), self.image.height())

    def initializeMitsuba(self):
        # Start up the scheduling system with one worker per local core
        self.scheduler = Scheduler.getInstance()
        for i in range(0, multiprocessing.cpu_count()):
            self.scheduler.registerWorker(LocalWorker(i, 'wrk%i' % i))
        self.scheduler.start()
        # Create a queue for tracking render jobs
        self.queue = RenderQueue()
        # Get a reference to the plugin manager
        self.pmgr = PluginManager.getInstance()

    def shutdownMitsuba(self):
        self.queue.join()
        self.scheduler.stop()

    def createScene(self):
        # Create a simple scene containing a sphere
```

```
        sphere = self.pmgr.createObject(Properties("sphere"))
        sphere.configure()
        scene = Scene()
        scene.addChild(sphere)
        scene.configure()
        # Don't automatically write an output bitmap file when the
        # rendering process finishes (want to control this from Python)
        scene.setDestinationFile('')
        return scene

    def render(self, scene):
        # Create a render job and insert it into the queue
        job = RenderJob('myRenderJob', scene, self.queue)
        job.start()
        # Wait for the job to finish
        self.queue.waitLeft(0)
        # Develop the camera's film into an 8 bit sRGB bitmap
        film = scene.getFilm()
        size = film.getSize()
        bitmap = Bitmap(Bitmap.ERGB, Bitmap.EUInt8, size)
        film.develop(Point2i(0, 0), size, Point2i(0, 0), bitmap)
        # Write to a PNG bitmap file
        outFile = FileStream("rendering.png", FileStream.ETruncReadWrite)
        bitmap.write(Bitmap.EPNG, outFile)
        outFile.close()
        # Also create a QImage (using a fast memory copy in C++)
        return QImage(bitmap.getNativeBuffer(),
            size.x, size.y, QImage.Format_RGB888)

    def paintEvent(self, event):
        painter = QPainter(self)
        painter.drawImage(QPoint(0, 0), self.image)
        painter.end()

def main():
    app = QApplication(sys.argv)
    view = MitsubaView()
    view.show()
    view.raise_()
    retval = app.exec_()
    view.shutdownMitsuba()
    sys.exit(retval)

if __name__ == '__main__':
    main()
```

13.2.11. Mitsuba interaction with PyQt/PySide (fancy)

The following snippet is a much fancier version of the previous PyQt/PySide example. Instead of waiting for the rendering to finish and then displaying it, this example launches the rendering in the background and uses Mitsuba's `RenderListener` interface to update the view and show image

blocks as they are being rendered. As before, some changes will be necessary to get this to run on PySide.



When using this snippet, please be wary of threading-related issues; the key thing to remember is that in Qt, only the main thread is allowed to modify Qt widgets. On the other hand, rendering and logging-related callbacks will be invoked from different Mitsuba-internal threads—this means that it's not possible to e.g. directly update the status bar message from the callback `finishJobEvent`. To do this, we must use use Qt's `QueuedConnection` to communicate this event to the main thread via signals and slots. See the code that updates the status and progress bar for more detail.

```
import mitsuba, multiprocessing, sys, time

from mitsuba.core import Scheduler, PluginManager, Thread, Vector, Point2i, \
    Vector2i, LocalWorker, Properties, Bitmap, Spectrum, Appender, EWarn, \
    Transform, FileStream
from mitsuba.render import RenderQueue, RenderJob, Scene, RenderListener
from PyQt4.QtCore import Qt, QPoint, QSize, QRect, pyqtSignal
from PyQt4.QtGui import QApplication, QMainWindow, QPainter, QImage, \
    QProgressBar, QWidget, QSizePolicy

Signal = pyqtSignal

class MitsubaRenderBuffer(RenderListener):
    """
    Implements the Mitsuba callback interface to capture notifications about
    rendering progress. Partially completed image blocks are efficiently
    tonemapped into a local 8-bit Mitsuba Bitmap instance and exposed as a QImage.
    """
    RENDERING_FINISHED  = 0
    RENDERING_CANCELLED = 1
    RENDERING_UPDATED   = 2
```

```
    GEOMETRY_CHANGED    = 3

    def __init__(self, queue, callback):
        super(MitsubaRenderBuffer, self).__init__()
        self.bitmap = self.qimage = None
        self.callback = callback
        self.time = 0
        self.size = Vector2i(0, 0)
        queue.registerListener(self)

    def workBeginEvent(self, job, wu, thr):
        """ Callback: a worker thread started rendering an image block.
            Draw a rectangle to highlight this """
        _ = self._get_film_ensure_initialized(job)
        self.bitmap.drawWorkUnit(wu.getOffset(), wu.getSize(), thr)
        self._potentially_send_update()

    def workEndEvent(self, job, wr, cancelled):
        """ Callback: a worker thread finished rendering an image block.
            Tonemap the associated pixels and store them in 'self.bitmap' """
        film = self._get_film_ensure_initialized(job)
        film.develop(wr.getOffset(), wr.getSize(), wr.getOffset(), self.bitmap)
        self._potentially_send_update()

    def refreshEvent(self, job):
        """ Callback: the entire image changed (some rendering techniques
            do this occasionally). Hence, tonemap the full film. """
        film = self._get_film_ensure_initialized(job)
        film.develop(Point2i(0), self.size, Point2i(0), self.bitmap)
        self._potentially_send_update()

    def finishJobEvent(self, job, cancelled):
        """ Callback: the rendering job has finished or was cancelled.
            Re-develop the image once more for the final result. """
        film = self._get_film_ensure_initialized(job)
        film.develop(Point2i(0), self.size, Point2i(0), self.bitmap)
        self.callback(MitsubaRenderBuffer.RENDERING_CANCELLED if cancelled
                      else MitsubaRenderBuffer.RENDERING_FINISHED)

    def _get_film_ensure_initialized(self, job):
        """ Ensure that all internal data structure are set up to deal
            with the given rendering job """
        film = job.getScene().getFilm()
        size = film.getSize()

        if self.size != size:
            self.size = size

            # Round the buffer size to the next power of 4 to ensure 32-bit
            # aligned scanlines in the underlying buffer. This is needed so
            # that QtGui.QImage and mitsuba.Bitmap have exactly the same
            # in-memory representation.
```

```
            bufsize = Vector2i((size.x + 3) // 4 * 4, (size.y + 3) // 4 * 4)

            # Create an 8-bit Mitsuba bitmap that will store tonemapped pixels
            self.bitmap = Bitmap(Bitmap.ERGB, Bitmap.EUInt8, bufsize)
            self.bitmap.clear()

            # Create a QImage that is backed by the Mitsuba Bitmap instance
            # (i.e. without doing unnecessary bitmap copy operations)
            self.qimage = QImage(self.bitmap.getNativeBuffer(), self.size.x,
                                 self.size.y, QImage.Format_RGB888)

            self.callback(MitsubaRenderBuffer.GEOMETRY_CHANGED)
        return film

    def _potentially_send_update(self):
        """ Send an update request to any attached widgets, but not too often """
        now = time.time()
        if now - self.time > .25:
            self.time = now
            self.callback(MitsubaRenderBuffer.RENDERING_UPDATED)

class RenderWidget(QWidget):
    """ This simple widget attaches itself to a Mitsuba RenderQueue instance
        and displays the progress of everything that's being rendered """
    renderingUpdated = Signal(int)

    def __init__(self, parent, queue, default_size = Vector2i(0, 0)):
        QWidget.__init__(self, parent)
        self.buffer = MitsubaRenderBuffer(queue, self.renderingUpdated.emit)
        # Need a queued conn. to avoid threading issues between Qt and Mitsuba
        self.renderingUpdated.connect(self._handle_update, Qt.QueuedConnection)
        self.setSizePolicy(QSizePolicy.Minimum, QSizePolicy.Minimum)
        self.default_size = default_size

    def sizeHint(self):
        size = self.buffer.size if not self.buffer.size.isZero() else self.
    default_size
        return QSize(size.x, size.y)

    def _handle_update(self, event):
        image = self.buffer.qimage
        # Detect when an image of different resolution is being rendered
        if image.width() > self.width() or image.height() > self.height():
            self.updateGeometry()
        self.repaint()

    def paintEvent(self, event):
        """ When there is more space then necessary, display the image centered
            on a black background, surrounded by a light gray border """
        QWidget.paintEvent(self, event)
        qp = QPainter(self)
        qp.fillRect(self.rect(), Qt.black)
```

```
        image = self.buffer.qimage
        if image is not None:
            offset = QPoint((self.width() - image.width()) / 2,
                (self.height() - image.height()) / 2)
            qp.setPen(Qt.lightGray)
            qp.drawRect(QRect(offset - QPoint(1, 1), image.size() + QSize(1, 1)))
            qp.drawImage(offset, image)
        qp.end()

class MitsubaDemo(QMainWindow):
    renderProgress = Signal(int)

    def __init__(self):
        super(MitsubaDemo, self).__init__()

        # Initialize Mitsuba
        self.initializeMitsuba()
        self.job = self.createRenderJob()
        self.job.setInteractive(True)

        # Initialize the user interface
        status = self.statusBar()
        self.rwidget = RenderWidget(self, self.queue,
            self.scene.getFilm().getSize())
        progress = QProgressBar(status)
        status.setContentsMargins(0,0,5,0)
        status.addPermanentWidget(progress)
        status.setSizeGripEnabled(False)
        self.setWindowTitle('Mitsuba/PyQt demo')
        self.setCentralWidget(self.rwidget)

        # Hide the scroll bar once the rendering is done
        def renderingUpdated(event):
            if event == MitsubaRenderBuffer.RENDERING_FINISHED:
                status.showMessage("Done.")
                progress.hide()

        self.renderProgress.connect(progress.setValue, Qt.QueuedConnection)
        self.rwidget.renderingUpdated.connect(renderingUpdated,
            Qt.QueuedConnection)

        # Start the rendering process
        status.showMessage("Rendering ..")
        self.job.start()

    def initializeMitsuba(self):
        # Start up the scheduling system with one worker per local core
        self.scheduler = Scheduler.getInstance()
        for i in range(0, multiprocessing.cpu_count()):
            self.scheduler.registerWorker(LocalWorker(i, 'wrk%i' % i))
        self.scheduler.start()
```

```
        # Create a queue for tracking render jobs
        self.queue = RenderQueue()
        # Get a reference to the plugin manager
        self.pmgr = PluginManager.getInstance()

        # Process Mitsuba log and progress messages within Python
        class CustomAppender(Appender):
            def append(self2, logLevel, message):
                print(message)
            def logProgress(self2, progress, name, formatted, eta):
                # Asynchronously notify the main thread
                self.renderProgress.emit(progress)

        logger = Thread.getThread().getLogger()
        logger.setLogLevel(EWarn) # Display warning & error messages
        logger.clearAppenders()
        logger.addAppender(CustomAppender())

    def closeEvent(self, e):
        self.job.cancel()
        self.queue.join()
        self.scheduler.stop()

    def createRenderJob(self):
        self.scene = self.pmgr.create({
            'type' : 'scene',
            'sphere' : {
                'type' : 'sphere',
            },
            'envmap' : {
                'type' : 'sunsky'
            },
            'sensor' : {
                'type' : 'perspective',
                'toWorld' : Transform.translate(Vector(0, 0, -5)),
                'sampler' : {
                    'type' : 'halton',
                    'sampleCount' : 64
                }
            }
        })

        return RenderJob('rjob', self.scene, self.queue)

    def keyPressEvent(self, e):
        if e.key() == Qt.Key_Escape:
            self.close()

def main():
    import signal
    # Stop the program upon Ctrl-C (SIGINT)
    signal.signal(signal.SIGINT, signal.SIG_DFL)
```

```
    app = QApplication(sys.argv)
    demo = MitsubaDemo()
    demo.show()
    demo.raise_()
    retval = app.exec_()
    sys.exit(retval)

if __name__ == '__main__':
    main()
```

13.2.12. Mitsuba interaction with NumPy

Suppose that `bitmap` contains a `mitsuba.core.Bitmap` instance (e.g. a rendering). Then the following snippet efficiently turns the image into a NumPy array:

```
import numpy as np
array = np.array(bitmap.getNativeBuffer())
```

14. Acknowledgments

I am indebted to my advisor Steve Marschner for allowing me to devote a significant amount of my research time to this project. His insightful and encouraging suggestions have helped transform this program into much more than I ever thought it would be.

The architecture of Mitsuba as well as some individual components are based on implementations discussed in: *Physically Based Rendering - From Theory To Implementation* by Matt Pharr and Greg Humphreys.

Some of the GUI icons were taken from the Humanity icon set by Canonical Ltd. The material test scene was created by Jonas Pilo, and the environment map it uses is courtesy of Bernhard Vogl.

The included index of refraction data files for conductors are copied from PBRT. They are originally from the Luxpop database (www.luxpop.com) and are based on data by Palik et al. [36] and measurements of atomic scattering factors made by the Center For X-Ray Optics (CXRO) at Berkeley and the Lawrence Livermore National Laboratory (LLNL).

The following people have kindly contributed code or bugfixes:

- Miloŝ Haŝan
- Marios Papas
- Edgar Velázquez-Armendáriz
- Jirka Vorba
- Leonhard Grünschloß

Mitsuba makes heavy use of the following amazing libraries and tools:

- Qt 4 by Digia
- OpenEXR by Industrial Light & Magic
- Xerces-C++ by the Apache Foundation
- Eigen by Benoît Jacob and Gaël Guennebaud
- SSE math functions by Julien Pommier
- The Boost C++ class library
- GLEW by Milan Ikits, Marcelo E. Magallon and Lev Povalahev
- Mersenne Twister by Makoto Matsumoto and Takuji Nishimura
- Cubature by Steven G. Johnson
- COLLADA DOM by Sony Computer Entertainment
- libjpeg-turbo by Darrell Commander and others
- libpng by Guy Eric Schalnat, Andreas Dilger, Glenn Randers-Pehrson and others
- libply by Ares Lagae

- BWToolkit by Brandon Walkin
- The SCons build system by the SCons Foundation

15. License

Mitsuba is licensed under the terms of Version 3 of the GNU General Public License, which is reproduced here in its entirety. The license itself is copyrighted © 2007 by the Free Software Foundation, Inc. `http://fsf.org/`.

15.1. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

15.2. Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work,

including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

   b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects

the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of

that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding

Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party

to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

18. End of Terms and Conditions / How to Apply These Terms to Your New Programs:

    If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

    To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

References

[1] Ashikhmin, M., and Shirley, P. An anisotropic phong BRDF model. *Graphics tools: The jgt editors' choice* (2005), 303.

[2] Blanco-Muriel, M., Alarcón-Padilla, D., López-Moratalla, T., and Lara-Coira, M. Computing the solar vector. *Solar Energy 70*, 5 (2001), 431–441.

[3] Blinn, J. F. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1978), SIGGRAPH '78, ACM, pp. 286–292.

[4] Bowers, J., Wang, R., Wei, L.-Y., and Maletz, D. Parallel poisson disk sampling with spectrum analysis on surfaces. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 166:1–166:10.

[5] Cline, D., Talbot, J., and Egbert, P. Energy redistribution path tracing. *ACM Trans. Graph. 24*, 3 (July 2005), 1186–1195.

[6] Cook, R. L., Halstead, J., Planck, M., and Ryu, D. Stochastic simplification of aggregate detail. In *ACM SIGGRAPH 2007 papers* (New York, NY, USA, 2007), SIGGRAPH '07, ACM.

[7] Dür, A. An Improved Normalization For The Ward Reflectance Model. *Journal of graphics, gpu, and game tools 11*, 1 (2006), 51–59.

[8] Eason, G., Veitch, A., Nisbet, R., and Turnbull, F. The theory of the back-scattering of light by blood. *Journal of Physics D: Applied Physics 11* (1978), 1463.

[9] Farrell, T., Patterson, M., and Wilson, B. A diffusion theory model of spatially resolved, steady-state diffuse reflectance for the noninvasive determination of tissue optical properties in vivo. *Medical physics 19* (1992), 879.

[10] Faure, H. Good permutations for extreme discrepancy. *Journal of Number Theory 42*, 1 (1992), 47–56.

[11] Geisler-Moroder, D., and Dür, A. A new ward brdf model with bounded albedo. In *Computer Graphics Forum* (2010), vol. 29, Wiley Online Library, pp. 1391–1398.

[12] Grünschloss, L., Raab, M., and Keller, A. Enumerating quasi-monte carlo point sequences in elementary intervals. *Monte Carlo and Quasi-Monte Carlo Methods* (2010).

[13] Hachisuka, T., and Jensen, H. W. Stochastic progressive photon mapping. *ACM Trans. Graph. 28*, 5 (Dec. 2009), 141:1–141:8.

[14] Hachisuka, T., Ogaki, S., and Jensen, H. W. Progressive photon mapping. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 130:1–130:8.

[15] Hanrahan, P., and Krueger, W. Reflection from layered surfaces due to subsurface scattering. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1993), SIGGRAPH '93, ACM, pp. 165–174.

[16] Henyey, L., and Greenstein, J. Diffuse radiation in the galaxy. *The Astrophysical Journal 93* (1941), 70–83.

[17] Irawan, P. *Appearance of woven cloth*. PhD thesis, Cornell University, Ithaca, NY, USA, 2008. http://ecommons.library.cornell.edu/handle/1813/8331.

[18] Jakob, W., Arbree, A., Moon, J., Bala, K., and Marschner, S. A radiative transfer framework for rendering materials with anisotropic structure. *ACM Transactions on Graphics (TOG), Proceedings of SIGGRAPH 2010 29*, 4 (2010), 53.

[19] Jakob, W., and Marschner, S. Manifold Exploration: a Markov Chain Monte Carlo technique for rendering scenes with difficult specular transport. *ACM Trans. Graph. 31*, 4 (July 2012), 58:1–58:13.

[20] Jarosz, W., Zwicker, M., and Jensen, H. W. The beam radiance estimate for volumetric photon mapping. *Computer Graphics Forum (Proceedings of Eurographics 2008) 27*, 2 (Apr. 2008), 557–566.

[21] Jensen, H. W. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96* (London, UK, UK, 1996), Springer-Verlag, pp. 21–30.

[22] Jensen, H. W., and Buhler, J. A rapid hierarchical rendering technique for translucent materials. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.

[23] Jensen, H. W., Marschner, S. R., Levoy, M., and Hanrahan, P. A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 511–518.

[24] Joe, S., and Kuo, F. Constructing sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing 30* (2008), 2635.

[25] Kajiya, J., and Kay, T. Rendering fur with three dimensional textures. *ACM Transactions on Graphics 23*, 3 (1989), 271–280.

[26] Kelemen, C., Szirmay-Kalos, L., Antal, G., and Csonka, F. A simple and robust mutation strategy for the metropolis light transport algorithm. In *Computer Graphics Forum* (2002), vol. 21, pp. 531–540.

[27] Keller, A. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 49–56.

[28] Kollig, T., and Keller, A. Efficient multidimensional sampling. *Computer Graphics Forum 21*, 3 (2002), 557–563.

[29] Křivánek, J., Bouatouch, K., Pattanaik, S. N., and Zára, J. Making radiance and irradiance caching practical: Adaptive caching and neighbor clamping. In *Proceedings of the Eurographics Symposium on Rendering Techniques, Nicosia, Cyprus, 2006* (2006), T. Akenine-Möller and W. Heidrich, Eds., Eurographics Association, pp. 127–138.

[30] Lafortune, E. P., and Willems, Y. D. Using the modified phong reflectance model for physically based rendering. Tech. rep., Cornell University, 1994.

[31] Lukáš Hošek and Alexander Wilkie. An analytic model for full spectral sky-dome radiance. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012) 31*, 4 (July 2012).

[32] Mitchell, D. P., and Netravali, A. N. Reconstruction filters in computer-graphics. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), SIGGRAPH '88, ACM, pp. 221–228.

[33] Narasimhan, S. G., Gupta, M., Donner, C., Ramamoorthi, R., Nayar, S. K., and Jensen, H. W. Acquiring scattering properties of participating media by dilution. *ACM Trans. Graph. 25*, 3 (July 2006), 1003–1012.

[34] Ngan, A., Durand, F., and Matusik, W. Experimental analysis of brdf models. In *Proceedings of the Eurographics Symposium on Rendering* (2005), vol. 2, Eurographics Association.

[35] Oren, M., and Nayar, S. Generalization of Lambert's reflectance model. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), ACM, pp. 239–246.

[36] Palik, E., and Ghosh, G. *Handbook of optical constants of solids*. Academic press, 1998.

[37] Phong, B.-T. Illumination for Computer Generated Pictures. *Communications of the ACM 18*, 6 (1975), 311–317.

[38] Preetham, A., Shirley, P., and Smits, B. A practical analytic model for daylight. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 91–100.

[39] Reinhard, E., Stark, M., Shirley, P., and Ferwerda, J. Photographic tone reproduction for digital images. *ACM Transactions on Graphics 21*, 3 (2002), 267–276.

[40] Saito, M., and Matsumoto, M. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. *Monte Carlo and Quasi-Monte Carlo Methods 2006* (2008), 607–622.

[41] Shirley, P., and Wang, C. Direct lighting calculation by monte carlo integration. In *In proceedings of the second EUROGRAPHICS workshop on rendering* (1991), pp. 54–59.

[42] Smits, B. An RGB-to-spectrum conversion for reflectances. *Graphics tools: The jgt editors' choice* (2005), 291.

[43] Tabellion, E., and Lamorlette, A. An approximate global illumination system for computer generated films. *ACM Trans. Graph. 23*, 3 (Aug. 2004), 469–476.

[44] Tevs, A., Ihrke, I., and Seidel, H.-P. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Symposium on Interactive 3D Graphics and Games (i3D'08)* (2008), pp. 183–190.

[45] Veach, E., and Guibas, L. Bidirectional estimators for light transport. In *Eurographics Rendering Workshop Proceedings* (1994).

[46] Veach, E., and Guibas, L. J. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 65–76.

[47] Walter, B. Notes on the ward brdf. Tech. Rep. PCG-05-06, Program of Computer Graphics, Cornell University, 2005.

[48] Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. Microfacet Models for Refraction through Rough Surfaces. *Rendering Techniques (Proceedings EG Symposium on Rendering)* (2007).

[49] Ward, G., and Heckbert, P. Irradiance gradients. In *Eurographics Rendering Workshop* (May 1992), pp. 85–98.

[50] Ward, G. J. Measuring and modeling anisotropic reflection. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), SIGGRAPH '92, ACM, pp. 265–272.

[51] Ward, G. J., Rubinstein, F. M., and Clear, R. D. A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph. 22*, 4 (June 1988), 85–92.

[52] Weidlich, A., and Wilkie, A. Arbitrarily layered micro-facet surfaces. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia* (New York, NY, USA, 2007), GRAPHITE '07, ACM, pp. 171–178.

[53] Yuksel, C., Kaldor, J. M., James, D. L., and Marschner, S. Stitch meshes for modeling knitted clothing with yarn-level detail. *ACM Trans. Graph. 31*, 4 (July 2012), 37:1–37:12.

[54] Zhao, S., Jakob, W., Marschner, S., and Bala, K. Building Volumetric Appearance Models of Fabric using Micro CT Imaging. *ACM Transactions on Graphics (TOG), Proceedings of SIGGRAPH 2011 30*, 4 (2011), 53.