Mitsuba Documentation

Version 0.3.1

Wenzel Jakob

April 13, 2012

Contents

Part I. Using Mitsuba

Disclaimer: This is 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 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. 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 *absolutely off-limits* to companies that develop rendering software not distributed under a compatible license.

3. Compiling the renderer

To compile Mitsuba, you will need a recent C++ compiler (e.g. GCC 4.1+ or Visual Studio 2008+) 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. There are some differences between the different operating systems—for more details, please refer to one of the next sections depending on which one you use.

3.1. Common steps

To get started, you will need to download a recent version of Mitsuba. Make sure that you have the Mercurial (http://mercurial.selenic.com/) versioning system installed[1] and enter the following at the command prompt:

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

Afterwards, you will need to download the precompiled dependencies into a new subdirectory named `mitsuba/dependencies`:

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

Common to all platforms is that a build configuration file must be chosen: amongst the following, please copy the best matching file into a new file to the root of the Mitsuba directory and rename it into `config.py`.

```
build/config-linux.py
build/config-darwin-x86_64.py
build/config-darwin-x86.py
build/config-darwin-universal.py
build/config-msvc2008-win32.py
build/config-msvc2008-win64.py
build/config-msvc2010-win32.py
build/config-msvc2010-win64.py
build/config-icl12-msvc2010-win32.py
build/config-icl12-msvc2010-win64.py
build/config-icl12-darwin-x86_64.py
build/config-icl12-darwin-x86.py
```

3.2. Compilation flags

Usually, you will not have to make any modification to this file, but sometimes a few minor edits may be necessary. In particular, you might want to add or remove certain compilation flags from the `CXXFLAGS` parameter. The following settings affect the behavior of Mitsuba:

`MTS_DEBUG` Enable assertions etc. Usually a good idea, and enabled by default.

`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.

[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.

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.

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 Forces the GUI to use a software fallback mode, which is considerably slower and removes the realtime preview. This is useful when running the interface on a remote Windows machine accessed via the Remote Desktop Protocol (RDP).

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`.

3.3. Building on Ubuntu Linux

You'll first need to install a number of dependencies. It is assumed here that you are using Ubuntu Linux (Maverick Meerkat / 10.10 or later), hence some of the package may be named differently if you are using another version.

First, run

```
$ sudo apt-get install build-essential scons mercurial qt4-dev-tools libpng12-dev
    libjpeg62-dev libilmbase-dev libxerces-c3-dev libboost1.42-all-dev
    libopenexr-dev libglewmx1.5-dev libxxf86vm-dev libpcrecpp0
    libboost-system-dev libboost-filesystem-dev libboost-python-dev libboost-dev
```

Please ensure that the installed version of the boost libraries is 1.42 or later. To get COLLADA support, you will also need to install the `collada-dom` packages or build it from scratch. Here, we install the `x86_64` binaries and development headers that can be found in the `dependencies/linux` directory[2]:

[2] The directory also contains source packages in case these binaries don't work for you.

```
$ sudo dpkg --install collada-dom_2.3.1-1_amd64.deb collada-dom-dev_2.3.1-1_amd64.
    deb
```

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 be able to run the renderer from the command line, you will first have to import it into your path:

```
$ . setpath.sh
```

(note the period at the beginning – this assumes that you are using `bash`). Having set up everything, you can now move on to Section 4.

3.3.1. Creating Ubuntu packages

For Ubuntu, the preferred way of redististributing executables is to create `.deb` package files. To create Mitsuba packages, it is strongly recommended that you work with a pristine Ubuntu installation[3]. This can be done as follows: first, install `debootstrap` and download the latest version of Ubuntu to a subdirectory (here, we use Maverick Meerkat, or version 10.10)

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

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

```
$ sudo chroot maverick-pristine
$ echo "deb http://archive.ubuntu.com/ubuntu maverick 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 3.3. 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.

[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.

3.3.2. Releasing Ubuntu packages

To redistribute Ubuntu packages over the Internet, 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> ./
```

into the `/etc/apt/sources.list` file.

3.4. 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

```
$ yum install mercurial gcc-c++ scons boost-devel qt4-devel OpenEXR-devel xerces-c-
    devel python-devel glew-devel collada-dom-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 be able to run the renderer from the command line, you will first have to import it into your path:

```
$ . setpath.sh
```

(note the period at the beginning – this assumes that you are using `bash`). Having set up everything, you can now move on to Section 4.

3.4.1. Creating Fedora Core packages

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

```
$ 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-0.3.1
$ tar czvf rpmbuild/SOURCES/mitsuba-0.3.1.tar.gz mitsuba-0.3.1/.
```

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

```
$ rpmbuild -bb mitsuba-0.3.1/data/linux/fedora/mitsuba.spec
```

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

3.5. 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>
```

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 be able to run the renderer from the command line, you will first have to import it into your path:

```
$ . setpath.sh
```

(note the period at the beginning – this assumes that you are using `bash`). Having set up everything, you can now move on to Section 4.

3.5.1. 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
```

3.6. Compiling on Gentoo Linux

Simon Haegler has contributed a Gentoo `ebuild` script that can be used to Mitsuba. It can be found inside the user repository `git://git.overlays.gentoo.org/user/mistafunk.git` and is located at `media-gfx/mitsuba/mitsuba-0.3.0.ebuild`.

3.7. Building on Windows

On the Windows platform, Mitsuba already includes many of the dependencies in precompiled form. There are a few things to set up though. First, please make sure that your installation of Visual Studio is up to date. In particular, for Visual Studio 2010, you'll need to apply Service Pack 1 or the resulting Mitsuba executables will crash (`http://www.microsoft.com/download/en/details.aspx?id=23691`).

Next, you will need to install Python 2.6.x [4] (`www.python.org`) and SCons [5] (`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.

Finally, you will need to build Qt 4.7 (or a newer version) from source, as pre-built binaries will unfortunately not work. At the time of this writing, a suitable source code package is available here: `http://get.qt.nokia.com/qt/source/qt-everywhere-opensource-src-4.7.4.zip`. It is important that the Qt build process is performed using the *exact same*[6] compiler that will also be used to build Mitsuba. This means that Qt should be compiled *after* applying any Visual Studio service packs. To configure Qt and start the compilation process, enter

```
C:\Qt>configure.exe -release -no-webkit -no-phonon -no-phonon-backend -no-script
      -no-scripttools -no-qt3support -no-multimedia -no-ltcg
..(configuration messages)..
C:\Qt>nmake
```

Having installed all dependencies, run the "Visual Studio *2008/2010* Command Prompt" from the Start Menu (`x86` for 32-bit or `x64` for 64bit). Afterwards, navigate to the Mitsuba directory. Depending on whether or not the Qt binaries are on the `%PATH%` environment variable, you might have to explicitly specify the Qt path:

```
C:\Mitsuba\>set QTDIR=C:\Qt
```

where `C:\Qt` is the path to your Qt installation. Afterwards, 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. 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.

3.7.1. Integration with the Visual Studio interface

Basic Visual Studio 2008 and 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-msvc2008.sln` and `build/mitsuba-msvc2010.sln`. These Visual Studio projects still internally use the SCons-based build system to compile Mitsuba; whatever build

[4] Make sure that you get a Python binary matching the architecture, for which you plan to compile Mitsuba (i.e. `x86` or `x86_64`) – this is needed by the Python bindings. If you wish to use another Python version, you will have to change `config.py` and supply your own Boost boost binaries linked against that version of Python.

[5] 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.

[6] The blame here goes to Microsoft for changing the C++ ABI whenever they feel like it.

configuration is selected within Visual Studio will be used to pick a matching configuration file from the `build` directory. Note that you will potentially have to add a `QTDIR="...`" line to each of the used configuration files when building directly from Visual Studio.

3.8. Building on Mac OS X

On Mac OS X, you will need to install both SCons (>2.0.0, available at `www.scons.org`) and a recent release of XCode. You will also need to get Qt 4.7.0 or a newer version — make sure that you get the normal Cocoa release (i.e. *not* the one based on Carbon). All of the other dependencies are already included in precompiled form. Now open a Terminal and 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 be able to run the renderer from the command line, you will have to import it into your path:

```
$ . setpath.sh
```

(note the period at the beginning – this assumes that you are using `bash`).

4. 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.

4.1. 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. A quick video tutorial on using the GUI can be found here: http://vimeo.com/13480342.

4.2. 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
```

It is also possible to connect to network render nodes, which essentially lets Mitsuba parallelize 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.

- **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

```
Mitsuba version 0.3.1, Copyright (c) 2012 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 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
```

4.2.1. Passing parameters

Any attribute in the XML-based scene description language 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

```xml
<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
```

4.2.2. 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.

4.2.3. Rendering an animation

The command line interface is ideally suited for rendering large amounts of 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 especially useful when parallelizing over multiple machines: as some of the participating 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
```

4.3. 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 4.2, 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.

4.4. 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.

5. 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="0.3.1">
    <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="0.3.1">
    <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 camera, a film, a sampler and one or more luminaires. Here is a more complex example:

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

<scene version="0.3.1">
    <integrator type="path">
        <!-- Path trace with a max. path length of 8 -->
        <integer name="maxDepth" value="8"/>
    </integrator>
```

```xml
    <!-- Instantiate a perspective camera with 45 degrees field of view -->
    <camera 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="exrfilm">
            <integer name="width" value="1920"/>
            <integer name="height" value="1080"/>
        </film>
    </camera>

    <!-- 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" x="-3" z="1"/>
        </transform>

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

This example introduces several new object types (`integrator`, `camera`, `bsdf`, `sampler`, `film`, and `luminaire`) 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 camera and how it records the resulting radiance samples, hence they are nested inside it.

5.1. 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.

5.1.1. 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.

5.1.2. Strings

Passing strings is straightforward:

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

5.1.3. Color spectra

Depending on the compilation flags of Mitsuba (see Section 3.2 for details), the renderer internally either represents colors using discretized color spectra (when SPECTRUM_SAMPLES is set to a value other than 3), or it uses a basic linear RGB representation[7]. Irrespective of which internal representation is used, Mitsuba supports several different ways of specifying color information, which is then converted appropriately.

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 completely uniform spectrum, it is to provide only a single value:

```
<spectrum name="spectrumProperty" 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 likely not work anymore 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 follows:

[7] The official releases all use linear RGB—to do spectral renderings, you will have to compile Mitsuba yourself.

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

Another convenient way of providing color spectra is by specifying linear RGB or sRGB values using floating-point triplets or hex values:

```
<rgb name="spectrumProperty" value="0.2, 0.8, 0.4"/>
<srgb name="spectrumProperty" value="0.4, 0.3, 0.2"/>
<srgb name="spectrumProperty" value="#f9aa34"/>
```

When Mitsuba is compiled with the default settings, it internally uses linear RGB to represent colors, so these values can directly be used. However, when configured for doing spectral rendering, a suitable color spectrum with the requested RGB reflectance must be found. This is a tricky problem, since there is an infinite number of spectra with this property.

Mitsuba uses a method by Smits et al. [20] to find a "plausible" spectrum that is as smooth as possible. To do so, it uses one of two methods depending on whether the spectrum contains a unitless reflectance value, or a radiance-valued intensity.

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

The `reflectance` intent is used by default, so remember to set it to `illuminant` when defining the brightness of a light source with the <rgb> tag.

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
...
```

Finally, it is also possible to specify the spectral distribution of a black body emitter, 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 luminaire introduces physical units into the rendering process of Mitsuba, which is ordinarily a unitless system[8].

Specifically, the black body spectrum has units of power (W) per unit area (m^{-2}) per steradian (sr^{-1}) per unit wavelength (nm^{-1}). Assuming that the scene is modeled in units of meters, the spectral power distribution of pixels rendered by a perspective camera will then have the exact same units (i.e. $W \cdot m^{-2} \cdot sr^{-1} \cdot nm^{-1}$).

If these units are inconsistent with your scene, you may use the optional `multiplier` attribute:

[8]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.

```xml
<!-- Oops, the scene is modeled in centimeters, not meters -->
<blackbody name="spectrumProperty" temperature="5000K" multiplier="0.01"/>
```

5.1.4. Vectors, Positions

Points and vectors can be specified as follows:

```xml
<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.

5.1.5. 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:

```xml
<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.

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

- Rotations around a specified direction. The angle is given in degrees, e.g.

  ```xml
  <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.

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

- Explicit 4×4 matrices, e.g

  ```xml
  <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.

  ```xml
  <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.

5.2. Instancing

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:

```
<scene version="0.3.1">
    <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[9]. 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.

5.3. 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"/>
```

[9] 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`").

6. Plugin reference

6.1. 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 6.2 for details). This BSDF characterizes what happens *at the surface*. In the XML scene description language, this might look like the following:

```
<scene version="0.3.1">
    <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="0.3.1">
    <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>
```

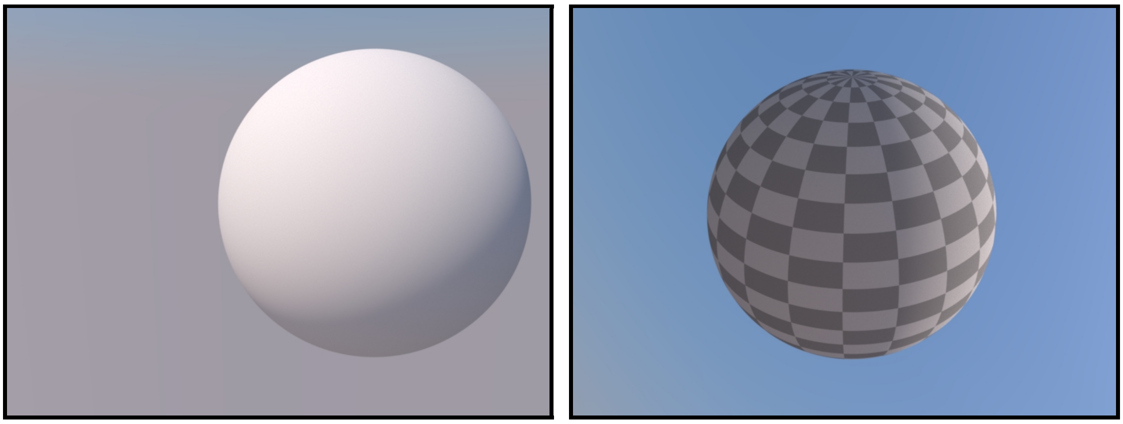
You may have noticed that the previous XML example dit not make any mention of surface scattering models (BSDFs). In Mitsuba, such a shape declaration creates an *index-matched* boundary. This means that incident illumination will pass through the surface without undergoing any kind of interaction. However, 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="0.3.1">
    <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>
```

6.1.1. 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	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)



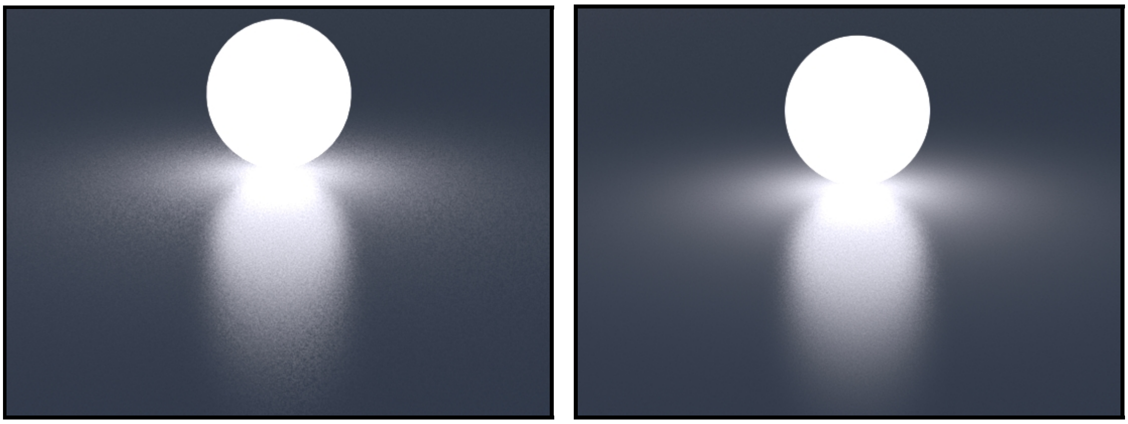
(a) Basic example, see Listing 2 (b) A textured sphere with the default parameterization

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

When using a sphere as the base object of an area luminaire, Mitsuba will switch to a special sphere luminaire sampling strategy [19] that works much better than the default approach. The resulting variance reduction makes it preferable to model most light sources as sphere luminaires (Figure 1).

```
<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 2: 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.



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

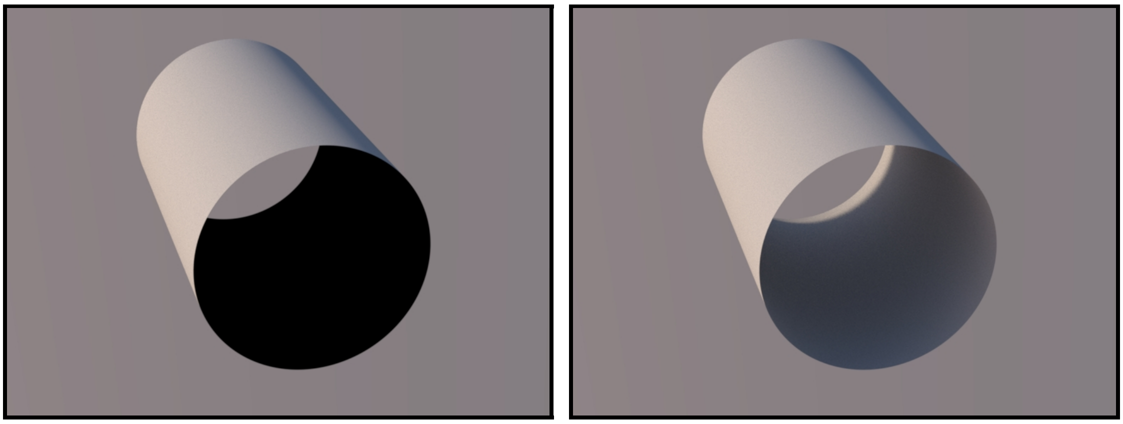
Figure 1: Area lights built from the combination of the area and sphere plugins produce renderings that have an overall lower variance.

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

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

Listing 3: Instantiation of a sphere luminaire

6.1.2. 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)
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))



(a) Cylinder with the default one-sided shading (b) Cylinder with two-sided shading, see Listing 4

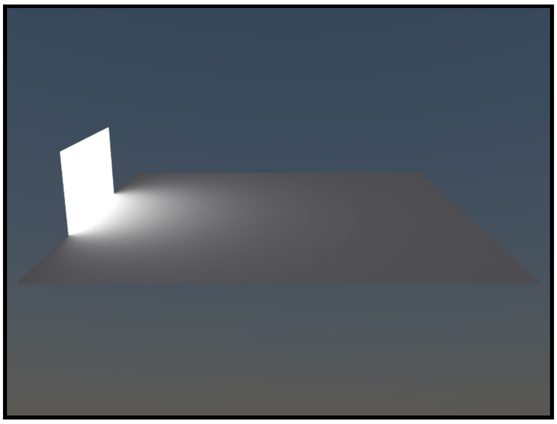
This shape plugin describes a simple cylinder intersection primitive. It should always be preferred over approximations modeled using triangles. Note that the cylinder does not have endcaps – also, it's interior has inward-facing normals, which most scattering models in Mitsuba will treat as fully absorbing. 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 4: A simple example for instantiating a cylinder, whose interior is visible

6.1.3. Rectangle intersection primitive (`rectangle`)

Parameter	Type	Description
`toWorld`	`transform`	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 5)

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.

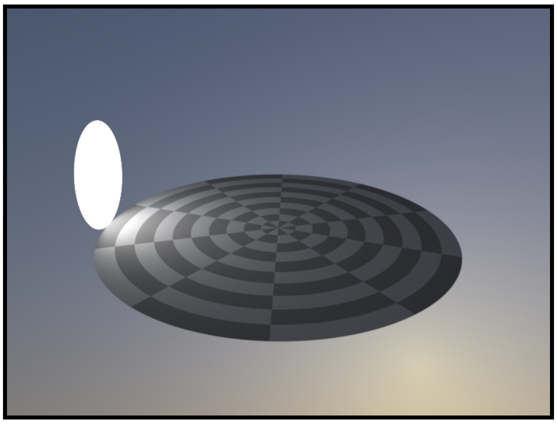
By default, the rectangle covers the XY-range $[-1, 1] \times [-1, 1]$ and has a surface normal that points into the positive Z direction. To change the rectangle scale, rotation, or translation, use the `toWorld` parameter.

```xml
<scene version="0.3.1">
    <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>
        <luminaire type="area">
            <spectrum name="intensity" value="3"/>
        </luminaire>
    </shape>
    <!-- ... other definitions ... -->
</scene>
```

Listing 5: A simple example involving two rectangle instances

6.1.4. Disk intersection primitive (disk)

Parameter	Type	Description
toWorld	transform	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 6)

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="0.3.1">
    <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>
        <luminaire type="area">
            <spectrum name="intensity" value="4"/>
        </luminaire>
    </shape>
</scene>
```

Listing 6: A simple example involving two disk instances

6.1.5. Wavefront OBJ mesh loader (`obj`)

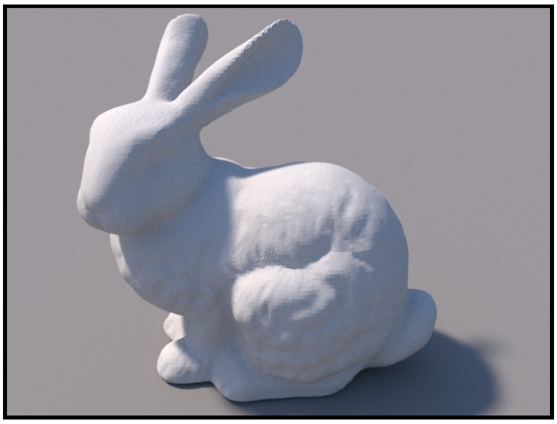
Parameter	Type	Description
`filename`	`string`	Filename of the OBJ file that should be loaded
`faceNormals`	`boolean`	When set to `true`, Mitsuba will use face normals when rendering the object, which will give it a faceted apperance. (Default: `false`)
`flipNormals`	`boolean`	Optional flag to flip all 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`	Specifies an optional linear object-to-world transformation. Note that non-uniform scales are not permitted! (Default: none (i.e. object space = world space))

This plugin implements a simple loader for Wavefront OBJ files. It handles triangle and quad meshes, vertex normals, and UV coordinates. Due to the heavy-weight nature of OBJ files, this loader is usually quite a bit slower than the `ply` or `serialized` plugins.

6.1.6. 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`)
`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. Note that non-uniform scales are not permitted! (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 heavy geometry. (Thai 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`.

6.1.7. 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. [3]). 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)

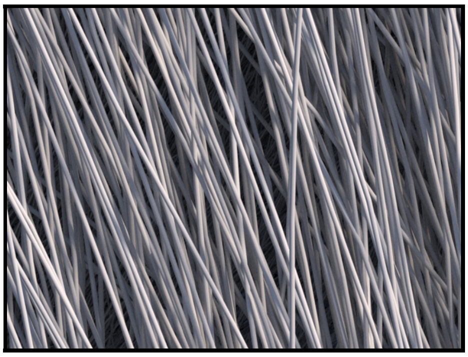


Figure 2: A close-up of the hair shape rendered with a diffuse scattering model (an actual hair scattering model will be needed for realistic apperance)

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.

6.1.8. 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.

Remarks:

- Note that it is currently 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 can currently not be used to replicate shapes with attached emitters, sensors, or subsurface integrators.

```xml
<!-- 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 7: An example of geometry instancing

6.1.9. Geometry instance (`instance`)

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

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

6.1.10. Animated geometry instance (`animatedinstance`)

Parameter	Type	Description
`filename`	`string`	Filename of an animated transformation
`(Nested plugin)`	`shapegroup`	A reference to a shape group that should be instantiated

This plugin implements an *animated* geometry instance, i.e. one or more shapes that are undergoing *ridgid* transformations over time.

The input file should contain a binary / serialized `AnimatedTransform` data structure – for details, please refer to the C++ implementation of this class.

6.1.11. Serialized mesh loader (`serialized`)

Parameter	Type	Description
`filename`	`string`	Filename of the gemoetry 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`)
`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. Note that non-uniform scales are not permitted! (Default: none (i.e. object space = world space))

This plugin represents the most space and time-efficient way of getting geometry into Mitsuba. It uses a highly efficient lossless compressed format for geometry storage. The format will be explained on this page in a subsequent revision of the documentation.

6.2. Surface scattering models

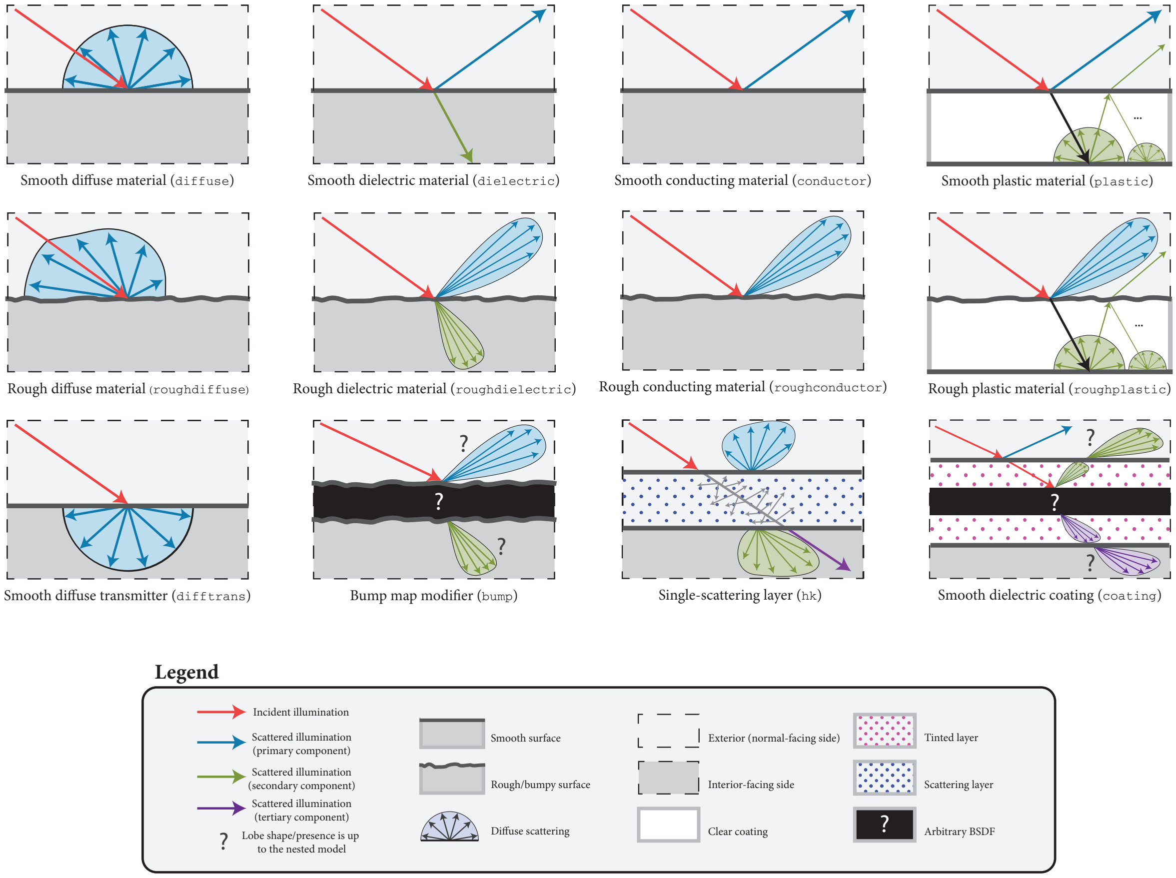


Figure 3: Schematic overview of the most important surface scattering models in Mitsuba (shown in the style of Weidlich and Wilkie [24]). The arrows indicate possible outcomes of an interaction with a surface that has the respective model applied to it.

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 6.5 and 6.4. 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.

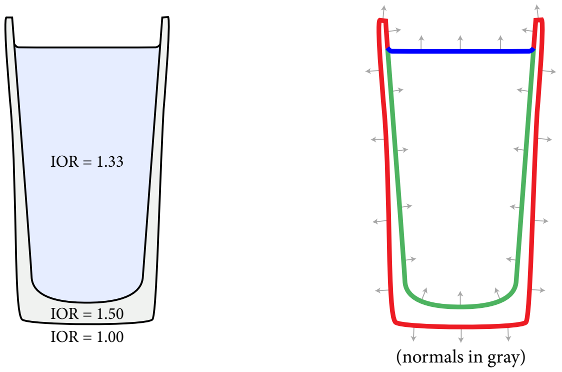
In Mitsuba, BSDFs are assigned to *shapes*, which describe the visible surfaces in the scene. In the scene description language, this assignment can either be performed by nesting BSDFs within shapes, or they can be named and then later referenced by their name. The following fragment shows an example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

```xml
<scene version="0.3.1">
    <!-- 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
(red)	1.50	1.00
(green)	1.50	1.33
(blue)	1.33	1.00

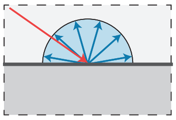
(c) Detailed IOR transitions

Figure 4: 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.

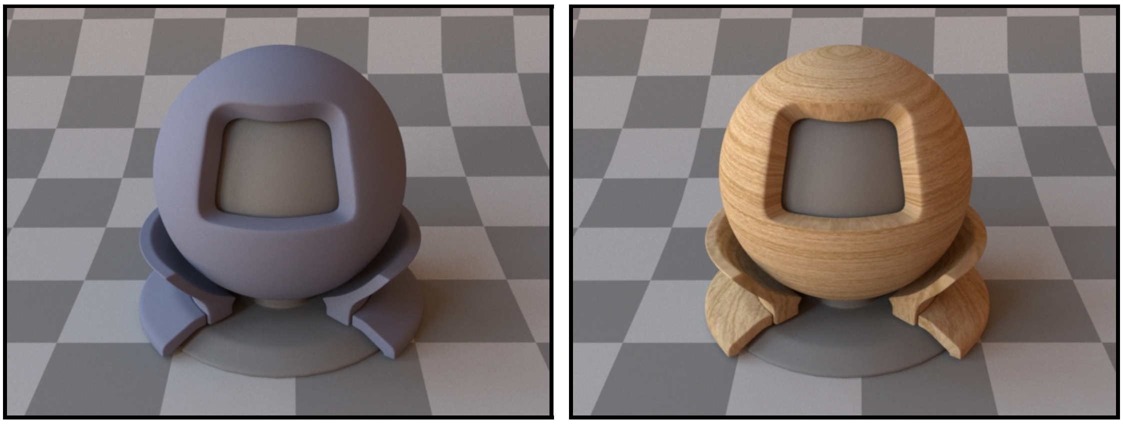
In many rendering systems, it would be feasible to increase the reflectance value above 1.0 in such a situation. But in Mitsuba, even a small surface that reflects a little more light than it receives will likely break the available rendering algorithms, or cause them to produce otherwise unpredictable results. In fact, we should rather change the lighting setup and then *reduce* the material's reflectance, since it is quite unlikely that we could find a real-world desk with a reflectance as high as 0.9.

As an example of the necessity for a meaningful material arrangement, consider the glass model illustrated in Figure 4. 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 some distance away from the glass.



6.2.1. 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 8 (b) Textured reflectance, see Listing 9

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.

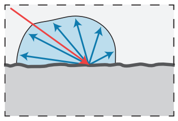
```
<bsdf type="diffuse">
    <srgb name="reflectance" value="#6d7185"/>
</bsdf>
```

Listing 8: A diffuse material, whose reflectance is specified as an sRGB color

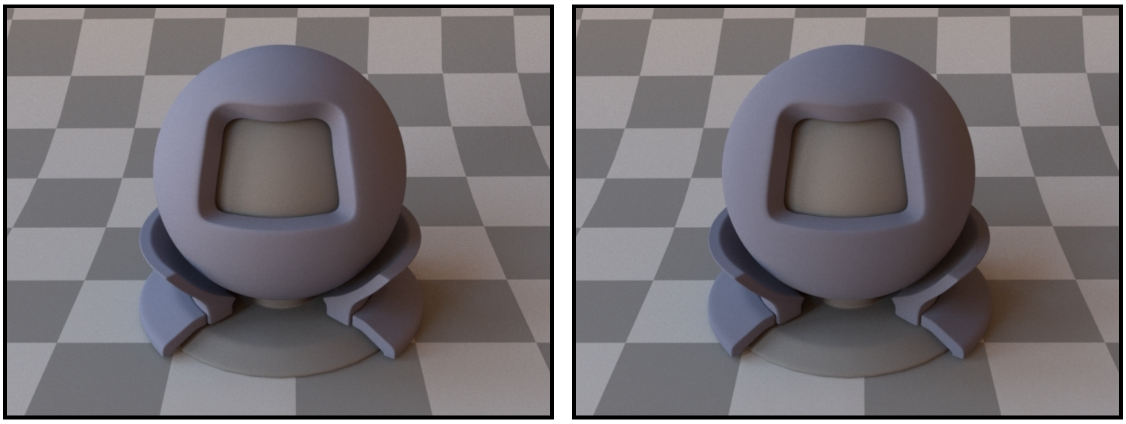
```
<bsdf type="diffuse">
    <texture type="bitmap" name="reflectance">
        <string name="filename" value="wood.jpg"/>
    </texture>
</bsdf>
```

Listing 9: A diffuse material with a texture map

6.2.2. 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 micro-geometry 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 5: 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 [14], 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.

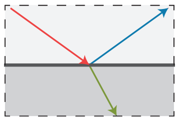
Since the original publication, this approach has been shown to be a good match for many real-world materials, particularly compared to Lambertian scattering, which does not take surface roughness into account.

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`).

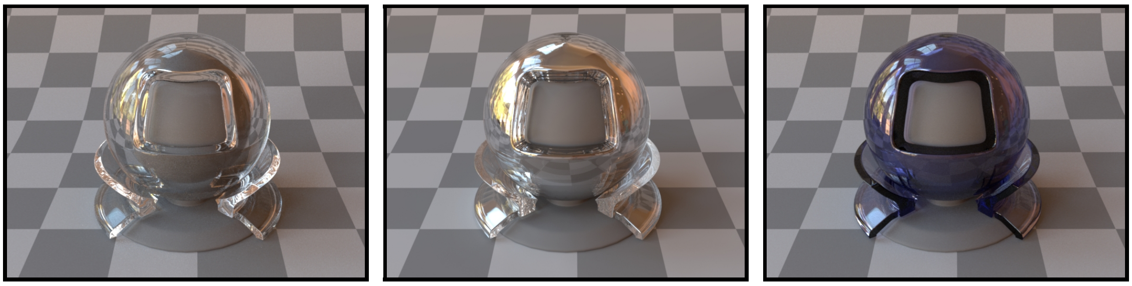
To get an intuition about the effect of the parameter α, consider the following approximate differentiation: a value of $\alpha = 0.001-0.01$ corresponds to a material with slight imperfections on an otherwise smooth surface (for such small values, the model will behave identically to `diffuse`), $\alpha = 0.1$ 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.

6.2.3. 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)
`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)
`specularTransmittance`	`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 10.
(b) Air↔Diamond (IOR: 2.419)
(c) Air↔Glass (IOR: 1.504) interface with absorption. See Listing 11.

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 10: 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 4 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.

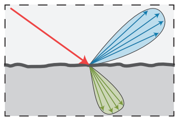
requires the use of a rendering technique that is aware of media (e.g. the volumetric path tracer). An example of how one might describe a slightly absorbing piece of glass is shown below:

```
<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 11: 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 1: 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.

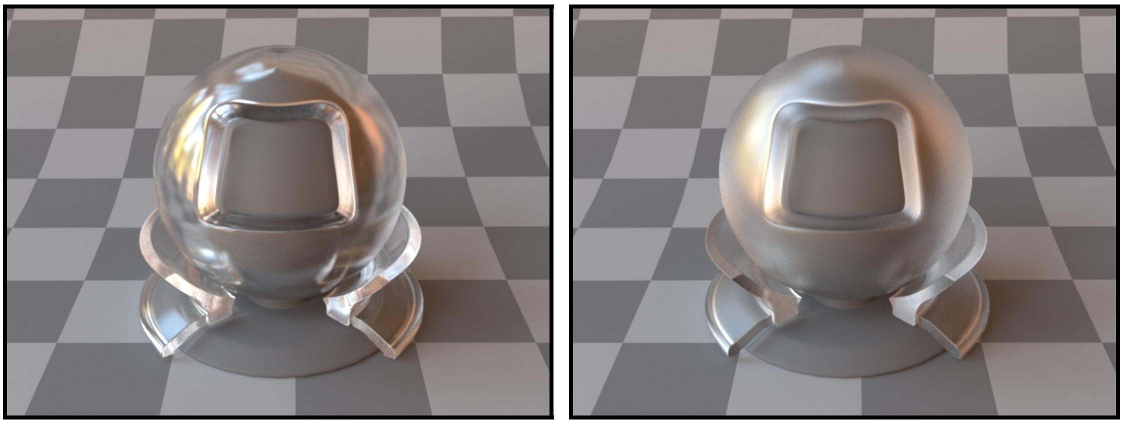


6.2.4. Rough dielectric material (`roughdielectric`)

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. [22], 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 micro-geometry. 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. [22]. 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 1 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.

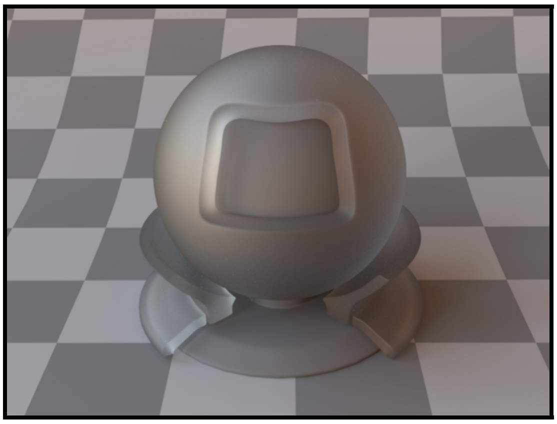
To get an intuition about the effect of the surface roughness parameter α, consider the following approximate differentiation: a value of $\alpha = 0.001 - 0.01$ corresponds to a material with slight imperfections on an otherwise smooth surface finish, $\alpha = 0.1$ is relatively rough, and $\alpha = 0.3 - 0.7$ 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 4 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.

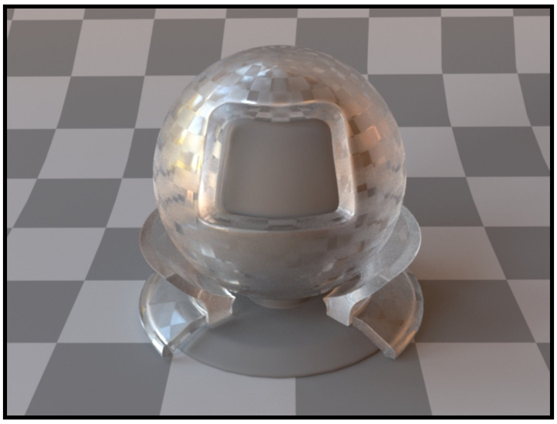
Technical details

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 12)



(b) Textured roughness (Listing 13)

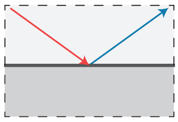
```
<bsdf type="roughdielectric">
    <string name="distribution" value="ggx"/>
    <float name="alpha" value="0.304"/>
    <string name="intIOR" value="bk7"/>
    <string name="extIOR" value="air"/>
</bsdf>
```

Listing 12: A material definition for ground glass

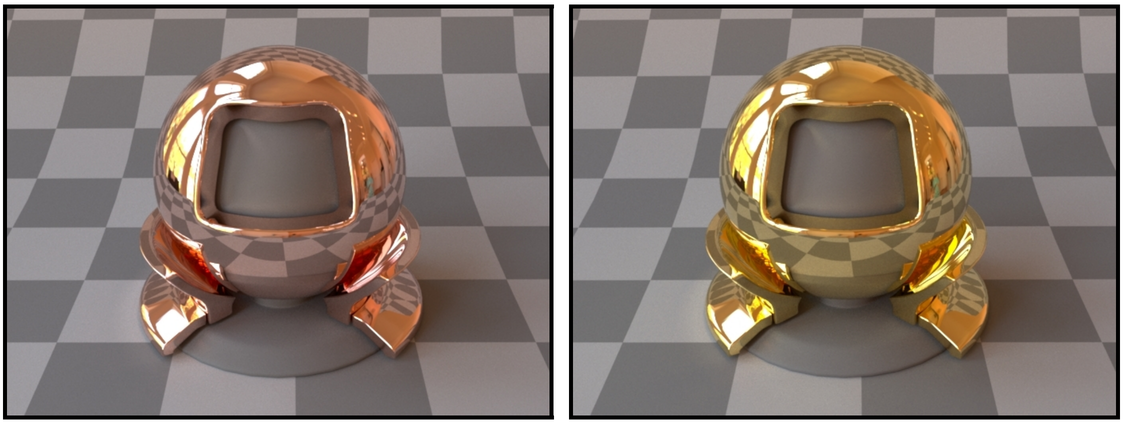
```
<bsdf type="roughdielectric">
    <string name="distribution" value="beckmann"/>
    <float name="intIOR" value="1.5046"/>
    <float name="extIOR" value="1.0"/>

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

Listing 13: A texture can be attached to the roughness parameter

6.2.5. Smooth conductor (conductor)



Parameter	Type	Description
material	string	Name of a material preset, see Table 2. (Default: Cu / copper)
eta	spectrum	Real part of the material's index of refraction (Default: based on the value of material)
k	spectrum	Imaginary part of the material's index of refraction, also known as absorption coefficient. (Default: based on the value of material)
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 14)

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 2 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. The table also contains a few birefingent materials, which are split into separate measurements corresponding to their two indices of refraction (named "ordinary" and "extraordinary ray").

When using this plugin, you should ideally compile Mitsuba with support for spectral rendering

[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. [15] and measurements of atomic scattering factors made by the Center For X-Ray Optics (CXRO) at Berkeley and the Lawrence Livermore National Laboratory (LLNL).

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 plugin.

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

Listing 14: 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 5.1.3 for details on the file format):

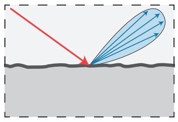
```
<shape type="...">
    <bsdf type="conductor">
        <spectrum name="eta" filename="conductorIOR.eta.spd"/>
        <spectrum name="k" filename="conductorIOR.k.spd"/>
    </bsdf>
<shape>
```

Listing 15: 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 (ord. ray)
Au	Gold	Se-e, Se-e_palik	Selenium (extr. ray)
Be, Be_palik	Polycrystalline beryllium	SiC, SiC_palik	Hexagonal silicon carbide
Cr	Chromium	SnTe, SnTe_palik	Tin telluride
CsI, CsI_palik	Cubic caesium iodide	Ta, Ta_palik	Tantalum
Cu, Cu_palik	Copper	Te, Te_palik	Trigonal tellurium (ord. ray)
Cu2O, Cu2O_palik	Copper (I) oxide	Te-e, Te-e_palik	Trigonal tellurium (extr. ray)
CuO, CuO_palik	Copper (II) oxide	ThF4, ThF4_palik	Polycryst. thorium (IV) fluoride
d-C, d-C_palik	Cubic diamond	TiC, TiC_palik	Polycrystalline titanium carbide
Hg, Hg_palik	Mercury	TiN, TiN_palik	Titanium nitride
HgTe, HgTe_palik	Mercury telluride	TiO2, TiO2_palik	Tetragonal titan. dioxide (ord. ray)
Ir, Ir_palik	Iridium	TiO2-e, TiO2-e_palik	Tetragonal titan. dioxide (extr. ray)
K, K_palik	Polycrystalline potassium	VC, VC_palik	Vanadium carbide
Li, Li_palik	Lithium	V_palik	Vanadium
MgO, MgO_palik	Magnesium oxide	VN, VN_palik	Vanadium nitride
Mo, Mo_palik	Molybdenum	W	Tungsten

Table 2: 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.

6.2.6. Rough conductor material (`roughconductor`)

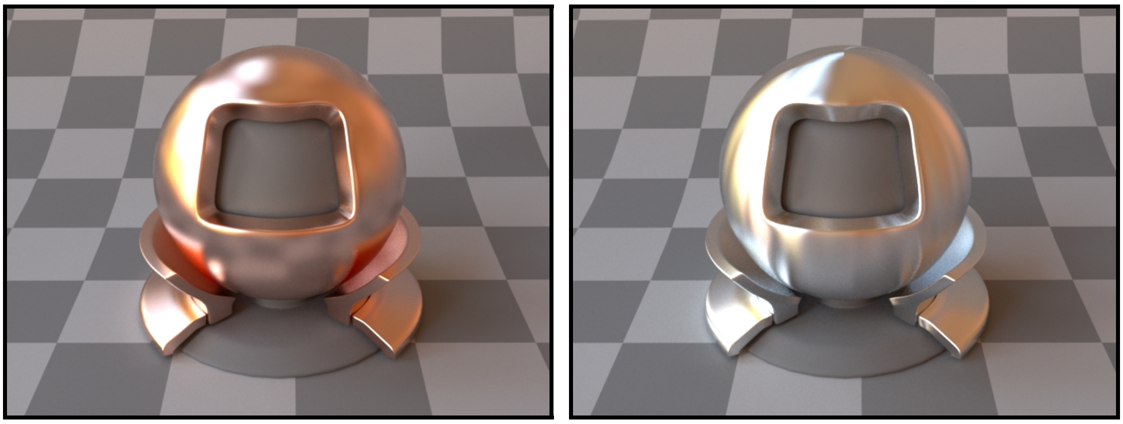


Parameter	Type	Description
`distribution`	`string`	Specifies the type of microfacet normal distribution used to model the surface roughness. (i) `beckmann`: Physically-based distribution derived from Gaussian random surfaces. This is the default. (ii) `ggx`: New distribution proposed by Walter et al. [22], 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 2. (Default: Cu / copper)
`eta`	`spectrum`	Real part of the material's index of refraction (Default: based on the value of `material`)
`k`	`spectrum`	Imaginary part of the material's index of refraction (the absorption coefficient). (Default: based on `material`)
`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)

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 empirical 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 16

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. [22]. 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 2 for the full list).

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 differentiation: 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>
```

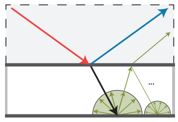
Listing 16: A material definition for brushed aluminium

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.

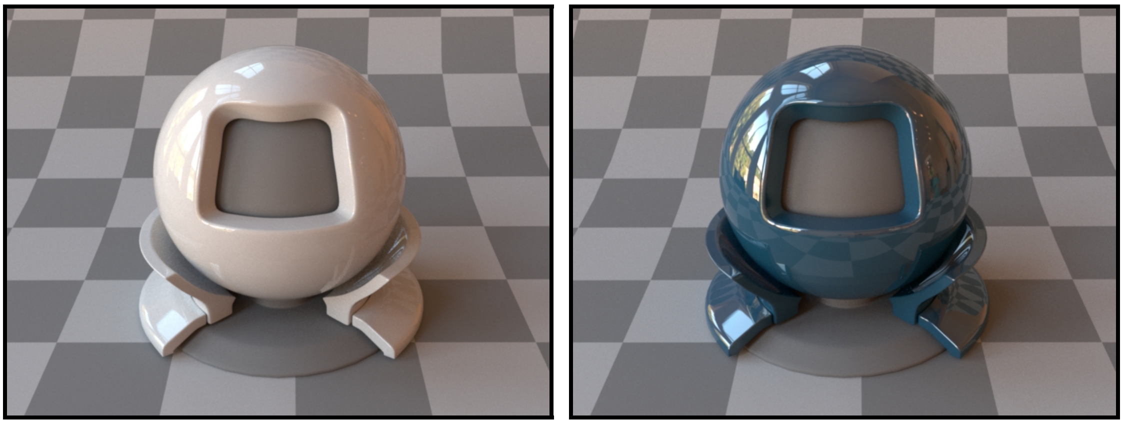
The Ashikhmin-Shirley microfacet distribution allows the specification of two distinct roughness values along the tangent and bitangent directions. This can be used to provide a material with a "brushed" appearance. The alignment of the anisotropy will follow the UV parameterization of the underlying 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.



6.2.7. 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 17)

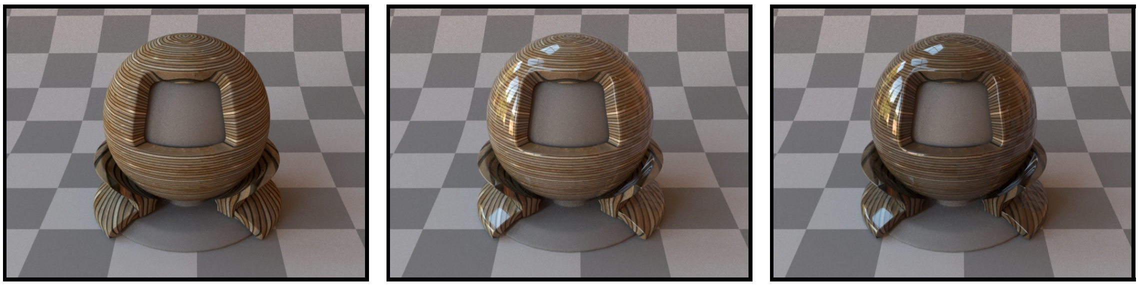
This plugin describes a smooth plastic-like material with internal scattering. It uses the Fresnel reflection and transmission coefficients to provide direction-dependent specular and diffuse components. Since it is simple, realistic, and fast, this model is often a better choice than the `phong`, `ward`, 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 1 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.

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

Listing 17: A shiny material whose diffuse reflectance is specified using sRGB

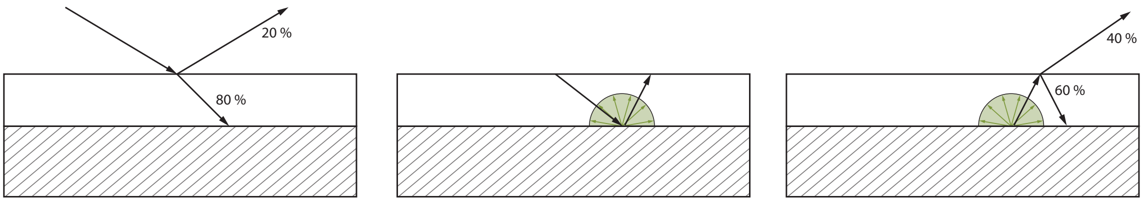


(a) Diffuse textured rendering (b) Plastic model, `nonlinear=false` (c) Plastic model, `nonlinear=true`

Figure 6: 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

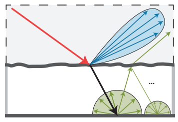
Figure 7: An illustration of the scattering events that are internally handled by this plugin

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 7a). The remaining illumination refracts into the material, where it scatters from the diffuse base layer. (Figure 7b). 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 7c).

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 6 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.

6.2.8. Rough plastic material (roughplastic)



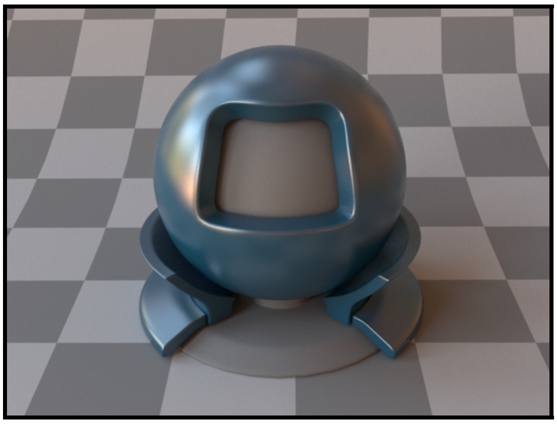
Parameter	Type	Description
distribution	string	Specifies the type of microfacet normal distribution used to model the surface roughness. (i) beckmann: Physically-based distribution derived from Gaussian random surfaces. This is the default. (ii) ggx: New distribution proposed by Walter et al. [22], which 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 empirical models like phong and ward when possible.

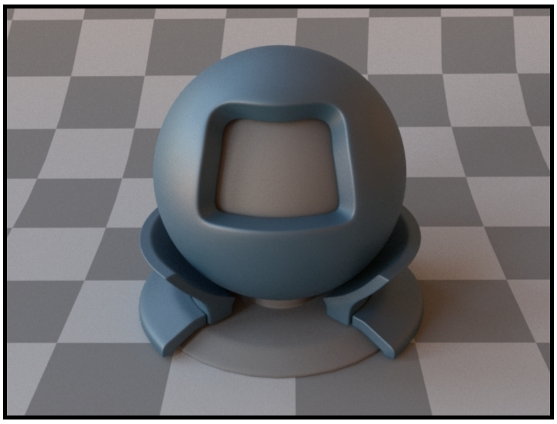
Microfacet theory describes rough surfaces as an arrangement of unresolved and ideally specular facets, whose normal directions are given by a specially chosen *microfacet distribution*. By accounting for shadowing and masking effects between these facets, it is possible to reproduce the important off-specular 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, $\alpha = 0.1$



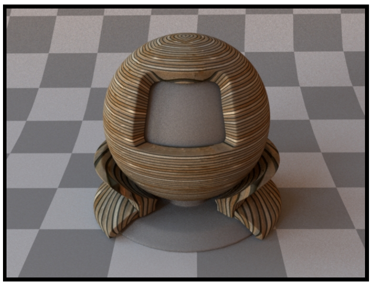
(b) GGX, $\alpha = 0.3$

known materials (see Table 1 on page 47 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.

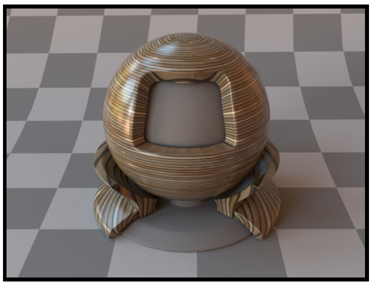
Like the `plastic` material, this model internally simulates the interaction of light with a diffuse base surface coated by a thin dielectric layer (where the coating layer is now *rough*). This is a convenient abstraction rather than a restriction. In other words, there are many materials that can be 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 56.

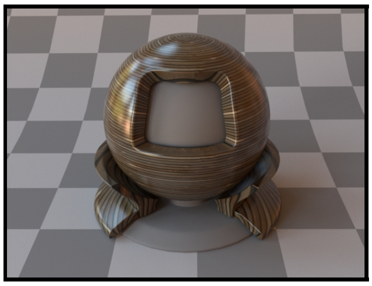
To get an intuition about the effect of the surface roughness parameter α, consider the following approximate differentiation: a value of $\alpha = 0.001 - 0.01$ corresponds to a material with slight imperfections on an otherwise smooth surface finish, $\alpha = 0.1$ is relatively rough, and $\alpha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



(a) Diffuse textured rendering

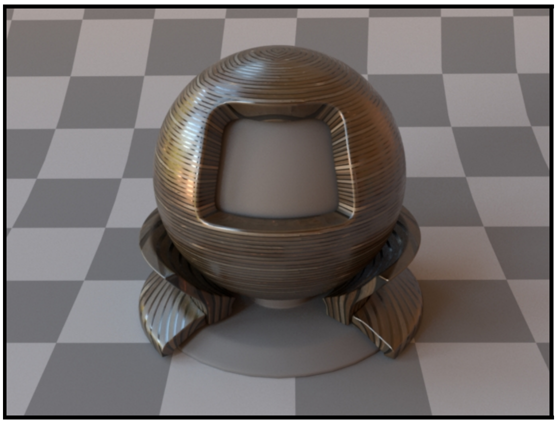


(b) Textured rough plastic model and `nonlinear=false`

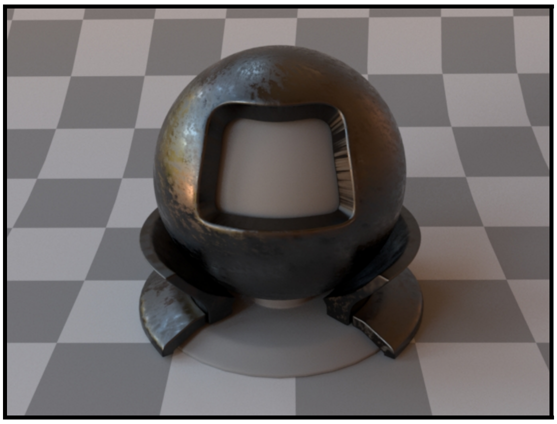


(c) Textured rough plastic model and `nonlinear=true`

Figure 8: 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 9: The ability to texture the roughness parameter makes it possible to render materials with a structured finish, as well as "smudgy" objects.

```xml
<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="bump.png"/>
        </texture>
        <float name="scale" value="0.6"/>
    </texture>
</bsdf>
```

Listing 18: A material definition for black plastic material with a spatially varying roughness.

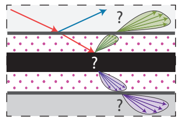
Technical details

The implementation of this model is partly based on the paper "Microfacet Models for Refraction through Rough Surfaces" by Walter et al. [22]. 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.

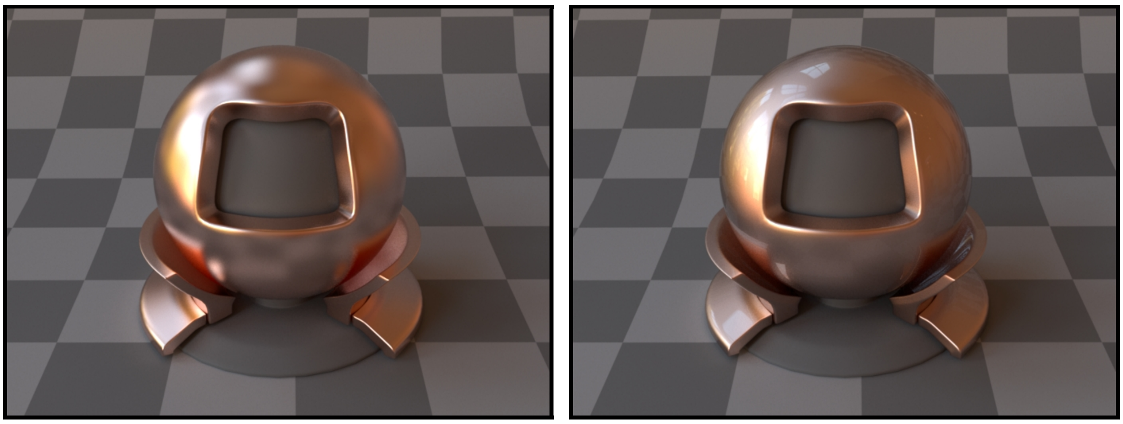
To avoid this performance issue, Mitsuba ships with data files (contained in the `data/microfacet` directory) containing precomputed values of this function over a large range of parameter values. At runtime, the relevant parts are extracted using tricubic interpolation.

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.



6.2.9. 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)
`specularTransmittance`	`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)
(Nested plugin)	`bsdf`	A nested BSDF model that should be coated.

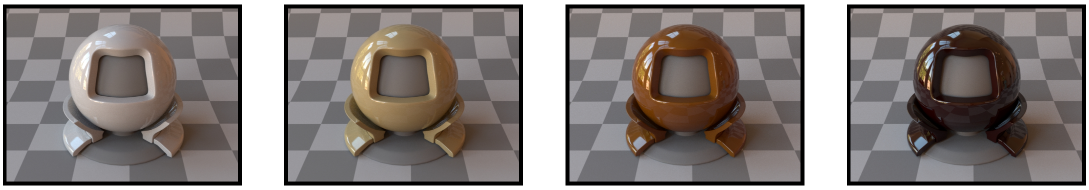


(a) Rough copper

(b) The same material coated with a single layer of clear varnish (see Listing 19)

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 [24]. 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.

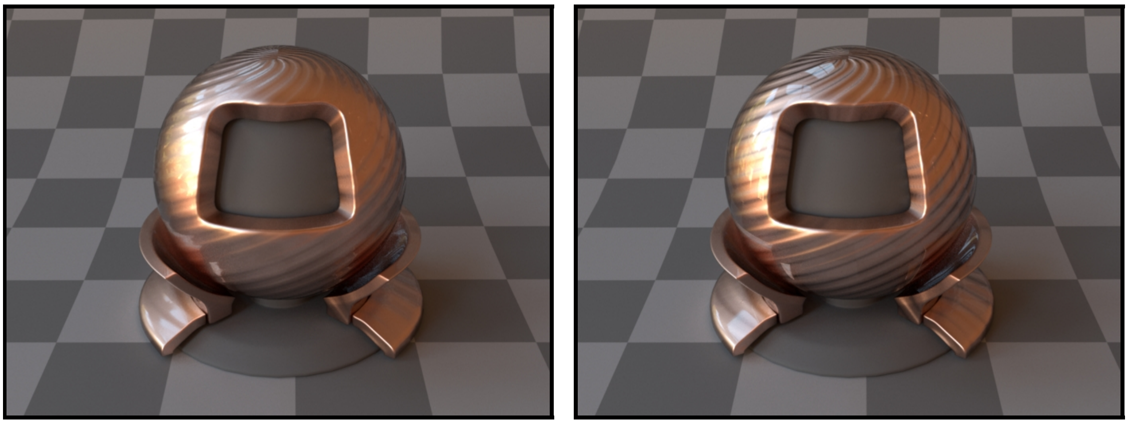


(a) thickness = 0 (b) thickness = 1 (c) thickness = 5 (d) thickness = 15

Figure 10: 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 19: Rough copper coated with a transparent layer of varnish



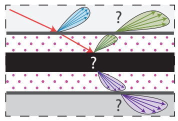
(a) Coated rough copper with a bump map applied on top (b) Bump mapped rough copper with a coating on top

Figure 11: Some interesting materials can be created simply by applying Mitsuba's material modifiers in different orders.

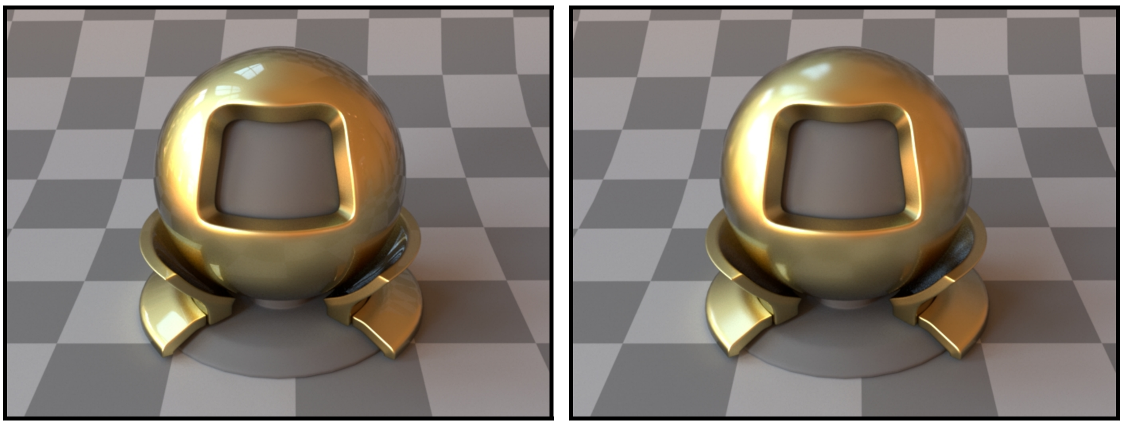
Technical details

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.

6.2.10. 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. [22], 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 micro-geometry. 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)
`sigmaA`	`spectrum` or `texture`	The absorption coefficient of the coating layer. (Default: 0, i.e. there is no absorption)
`specularTransmittance`	`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)
(Nested plugin)	`bsdf`	A nested BSDF model that should be coated.



(a) Rough gold coated with a *smooth* varnish layer

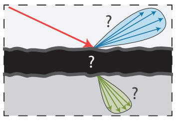
(b) Rough gold coated with a *rough* ($\alpha = 0.03$) varnish layer

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

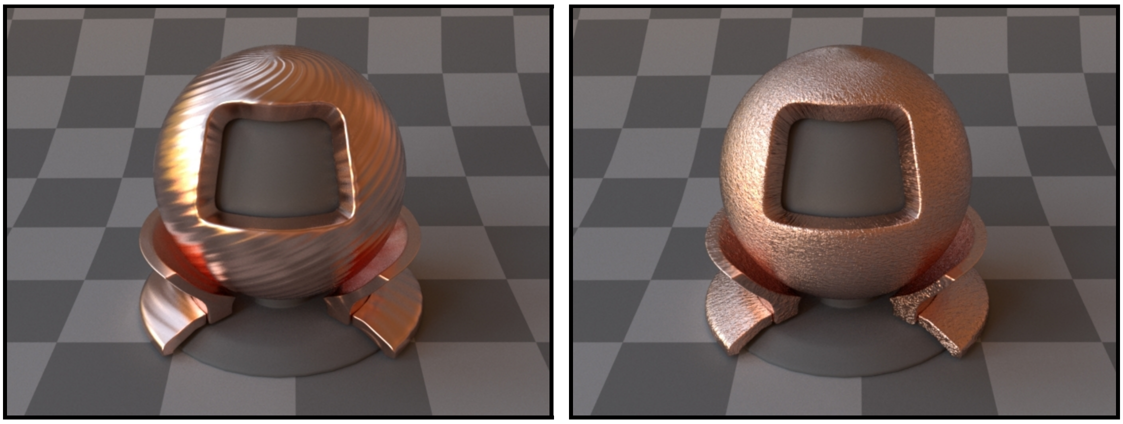
The implementation here is influenced by the paper "Arbitrarily Layered Micro-Facet Surfaces" by Weidlich and Wilkie [24].

[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.



6.2.11. Bump map modifier (**bump**)

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 [2] 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.

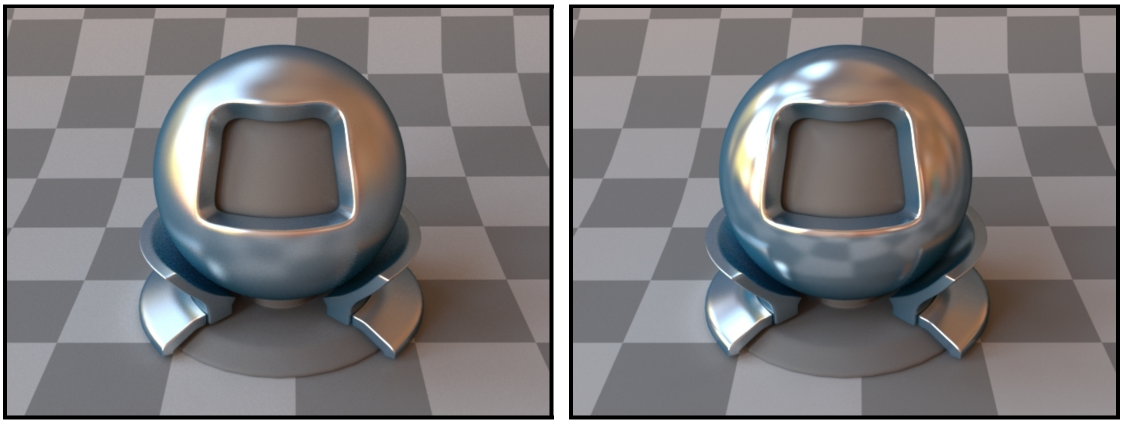
Note that the magnitude of the height field variations influences the strength of the displacement. If desired, the `scale` texture plugin can be used to magnify or reduce the effect of a bump map texture.

```
<bsdf type="bump">
    <!-- 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 20: A rough metal model with a scaled image-based bump map

6.2.12. 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

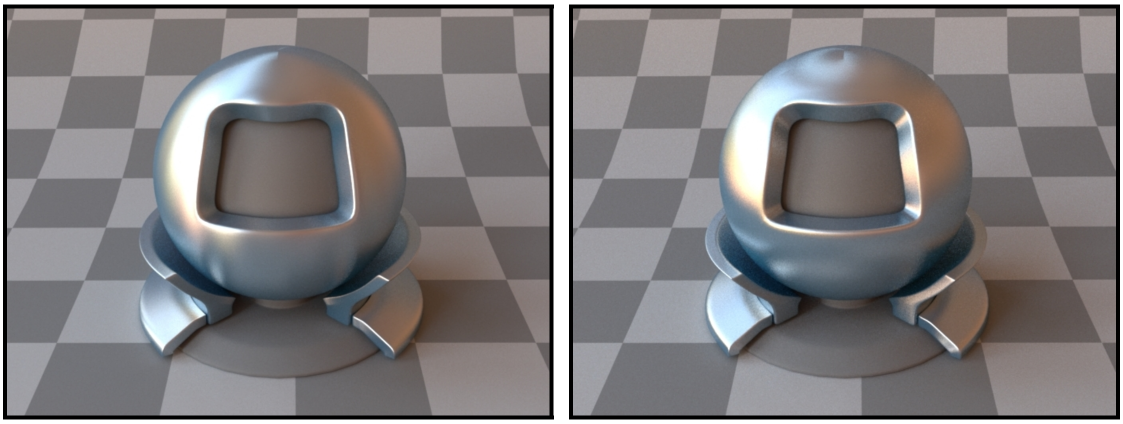
This plugin implements the modified Phong reflectance model as described in [16] and [12]. This empirical model is mainly included for historical reasons—its use in new scenes is discouraged, since significantly more realistic models have been developed since 1975.

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.

6.2.13. 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 [23] — suffers from energy loss at grazing angles. (ii) `ward-duer`: Corrected Ward model with lower energy loss at grazing angles [4]. Does not always conserve energy. (iii) `balanced`: Improved version of the `ward-duer` model with energy balance at all angles [5].
`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 [23]

(ii) "Notes on the Ward BRDF" by Bruce Walter [21]

(iii) "An Improved Normalization for the Ward Reflectance Model" by Arne Dür [4]

(iv) "A New Ward BRDF Model with Bounded Albedo" by Geisler-Moroder et al. [5]

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. [13], the Ward model performed noticeably worse than models based on microfacets.

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).

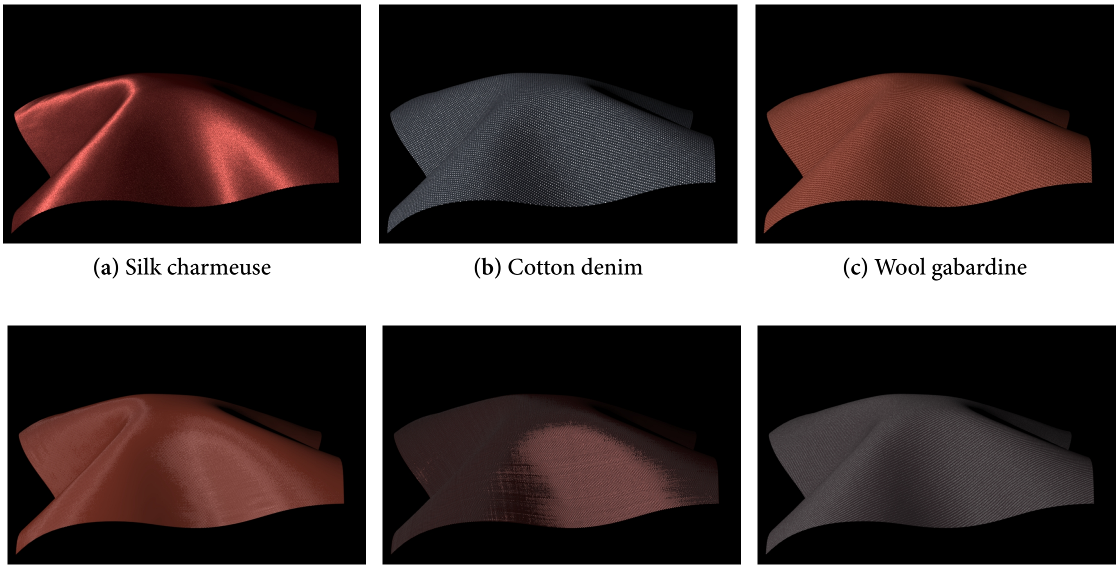
When using this plugin, note that the diffuse and specular reflectance components should add up to a value less than or equal to one (for each color channel). Otherwise, they will automatically be scaled appropriately to ensure energy conservation.

6.2.14. 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
`ksFactor`	`float`	Multiplicative factor of the specular component
`kdFactor`	`float`	Multiplicative factor of the diffuse component

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" [8]). 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.

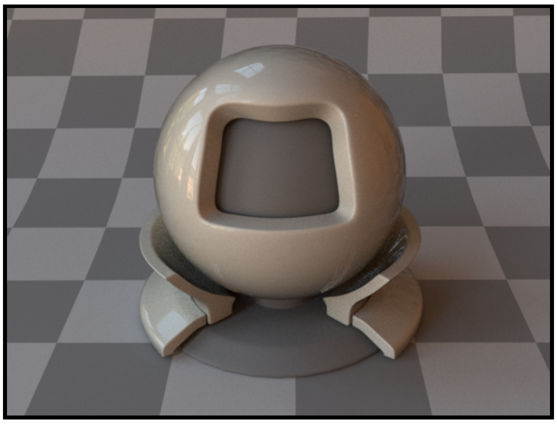


(a) Silk charmeuse (b) Cotton denim (c) Wool gabardine

(d) Polyester lining cloth (e) Silk shantung (f) Cotton twill

6.2.15. Random medium BRDF (rmbrdf)

Parameter	Type	Description
material	string	Name of a material preset, see Table 3. (Default: skin1)
sigmaS	spectrum or texture	Specifies the scattering coefficient of the layer. (Default: based on material)
sigmaA	spectrum or texture	Specifies the absorption coefficient of the 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)
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: air / 1.000277)
g	float or string	Specifies the phase function anisotropy — see the hg plugin for details (Default: 0, i.e. isotropic)
alpha	float or texture	Specifies the roughness of the unresolved surface micro-geometry. (Default: 0.1, i.e. the surface has a slightly rough finish)



(a) Rendering using the whole milk material preset

This plugin implements a BRDF scattering model that emulates interactions with a random medium embedded inside a dielectric layer. By approximating these events using a BRDF, any scattered illumination is assumed to exit the material *directly* at the original point of incidence. To simulate actual subsurface scattering, refer to Sections 6.5 and 6.4.

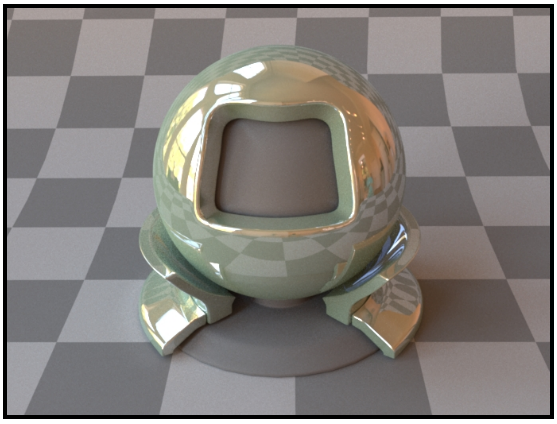
Note that renderings with this BRDF will usually look very similar to what might also be obtained using plastic. The plugin's reason for existance is that can be configured using parameters that are traditionally reserved for participating media.

Implementation details

Internally, the model is implemented by instantiating a Hanrahan-Krueger BSDF for single scattering in an infinitely thick layer together with an approximate multiple scattering component based on Jensen's [10] integrated dipole BRDF. These are then embedded into a dielectric layer using either the `coating` or `roughcoating` plugins depending on whether or not `alpha`=0. This yields a very convenient parameterization of a scattering model that behaves similarly to a coated diffuse material, but expressed in terms of the scattering and absorption coefficients `sigmaS` and `sigmaA`.

6.2.16. 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



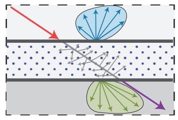
(a) An admittedly not particularly realistic linear combination of diffuse and specular BSDFs (Listing 21)

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.

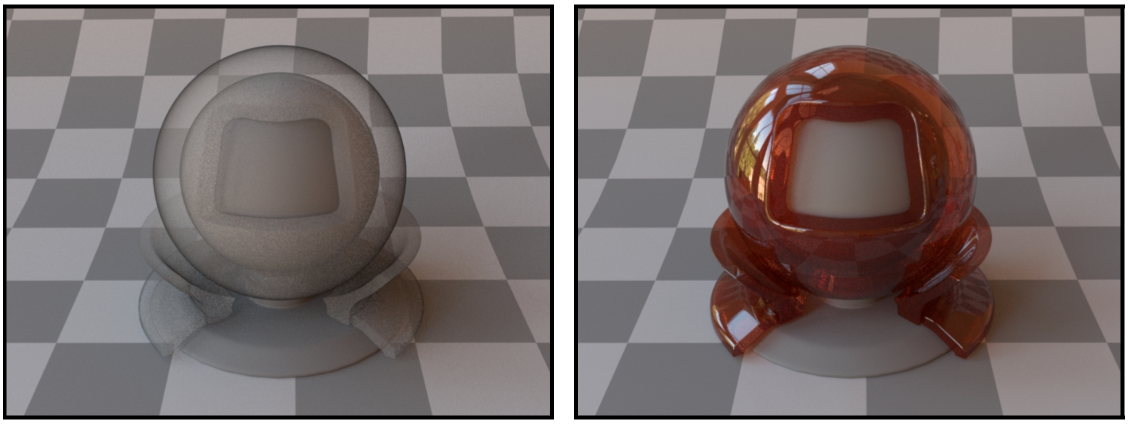
```xml
<bsdf type="mixturebsdf">
    <string name="weights" value="0.7, 0.2"/>

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

    <bsdf type="roughdiffuse">
        <rgb name="reflectance" value=".7 1 .7"/>
        <float name="alpha" value="0.4"/>
    </bsdf>
</bsdf>
```

Listing 21: A material definition for a mixture of 70% smooth chromium, 20% of a greenish rough diffuse material (and 10% absorption)

6.2.17. Hanrahan-Krueger BSDF (hk)



Parameter	Type	Description
material	string	Name of a material preset, see Table 3. (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 22)

Figure 12: Renderings using the uncoated and coated form of the Hanrahan-Krueger model.

This plugin provides an implementation of the Hanrahan-Krueger BSDF [6] 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:

```xml
<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 [6].

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 12. A solution is to use the sssbrdf plugin, which adds an approximate multiple scattering component.

```xml
<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>
```

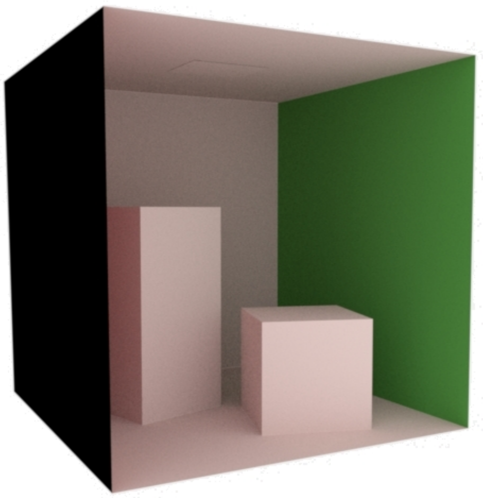
Listing 22: A thin dielectric layer with measured ketchup scattering parameters

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

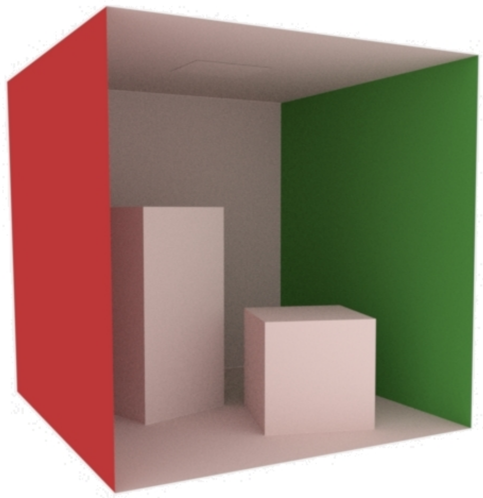
The implementation in Mitsuba is based on code by Tom Kazimiers and Marios Papas. Marios Papas has kindly verified the implementation of the coated and uncoated variants against both a path tracer and a separate reference implementation.

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

Parameter	Type	Description
(Nested plugin)	`bsdf`	A nested BRDF that should be turned into a two-sided scattering model.



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



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

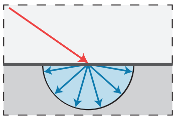
By default, all non-transmissive scattering models in Mitsuba are *one-sided* — in other words, they absorb all light that is received on the interior-facing side of any associated surfaces. Holes and visible back-facing parts are thus exposed as black regions.

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.

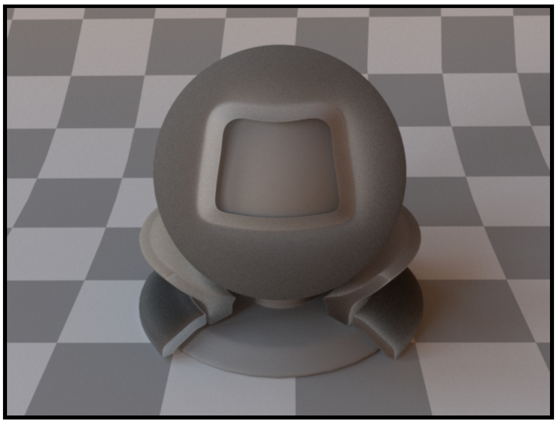
```
<bsdf type="twosided">
    <bsdf type="diffuse">
        <spectrum name="reflectance" value="0.4"/>
    </bsdf>
</bsdf>
```

Listing 23: A two-sided diffuse material

6.2.19. Diffuse transmitter (difftrans)



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



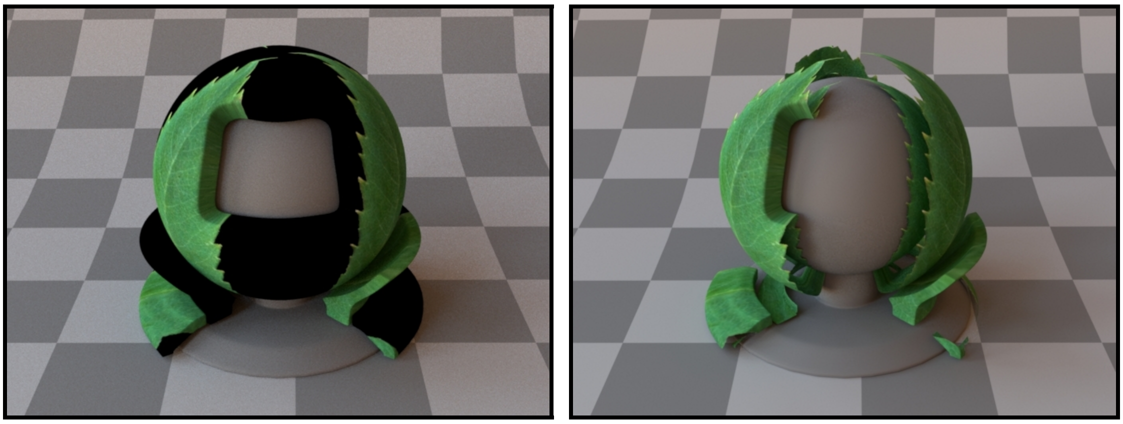
(a) The model with default parameters

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.

6.2.20. 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 24)

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 implemented as a forward-facing Dirac delta distribution.

```xml
<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.jpg"/>
            </texture>
        </bsdf>
    </bsdf>

    <!-- Fetch the opacity mask from a bitmap -->
    <texture name="opacity" type="bitmap">
        <string name="filename" value="leaf_opacity.jpg"/>
        <float name="gamma" value="1"/>
    </texture>
</bsdf>
```

Listing 24: Material configuration for a transparent leaf

6.2.21. 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

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

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.

6.3. Textures

The following section describes the available texture sources. In Mitsuba, textures are objects that can be attached to scattering model parameters supporting the "`texture`" type (see Section 6.2 for examples).

6.3.1. 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.

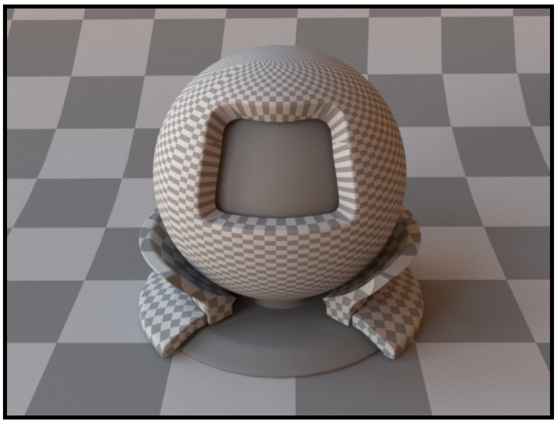
Here is an example:

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

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

Listing 25: Rendering a PLY file with vertex colors

6.3.2. Checkerboard (`checkerboard`)

Parameter	Type	Description
`color0, color1`	`spectrum`	Color values for the two differently-colored patches (Default: 0.4 and 0.2)
`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) Checkerboard applied to the material test object

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

6.3.3. Bitmap texture (`bitmap`)

Parameter	Type	Description
`filename`	`string`	Filename of the bitmap to be loaded
`gamma`	`float`	Gamma value of the source bitmap file (Default: *automatic*, i.e. linear for EXR input, and sRGB for everything else.)
`filterType`	`string`	Specifies the texture filturing 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) `none`: No filtering, do nearest neighbor lookups. Default: `ewa`.
`wrapMode`	`string`	This parameter defines the behavior of the texture outside of the $[0,1]$ uv range. (i) `repeat`: Repeat the texture (i.e. uv coordinates are taken modulo 2) (ii) `clamp`: Clamp uv coordinates to $[0,1]$ (iii) `black`: Switch to a zero-valued texture (iv) `white`: Switch to a one-valued texture Default: `repeat`.
`maxAnisotropy`	`float`	Specifies an upper limit on the amount of anisotropy of `ewa` lookups (Default: 8)
`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

This plugin implements a bitmap-based texture, which supports the following file formats:

- OpenEXR
- JPEG
- PNG (Portable Network Graphics)
- TGA (Targa)
- BMP (Windows bitmaps)

The plugin internally converts all bitmap data into a *linear* space to ensure a proper workflow.

6.3.4. 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

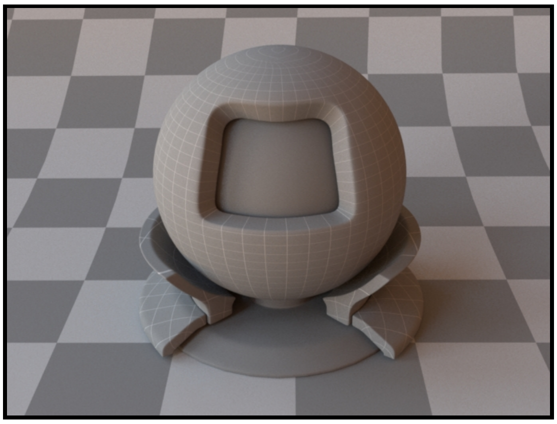
This simple plugin wraps a nested texture plugin and multiplies its contents by a user-specified value. This can be quite useful when a texture is too dark or too bright. The plugin can also be used to adjust the height of a bump map when using the bump plugin.

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

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

Listing 26: Scaling the contents of a bitmap texture

6.3.5. 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

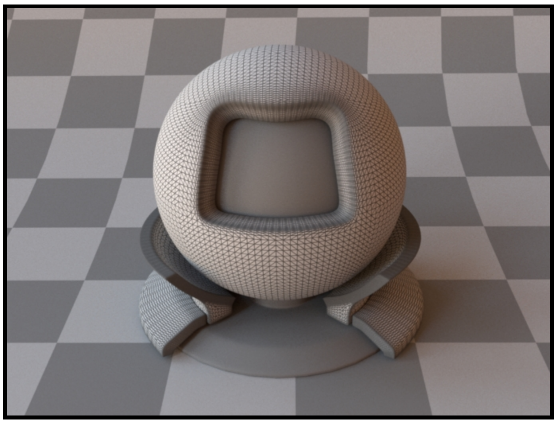


(a) Grid texture applied to the material test object

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

6.3.6. Wireframe texture (wireframe)

Parameter	Type	Description
interiorColor	spectrum	Color value of the interior of triangles (Default: 0.5)
color1	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.

6.4. Subsurface scattering

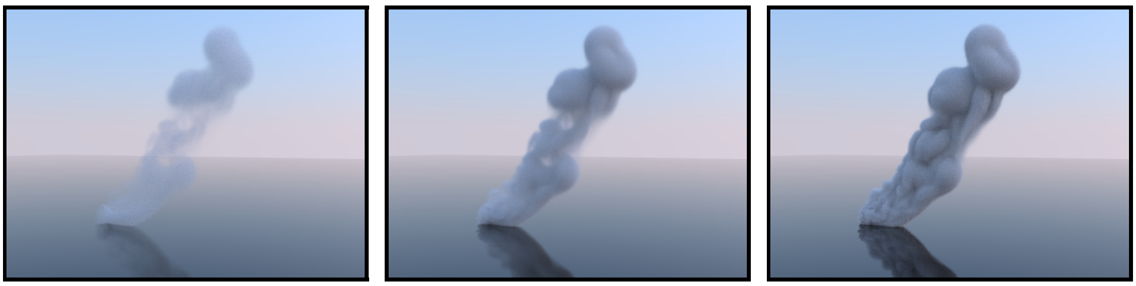
TBD

6.5. Participating media

TBD

6.5.1. 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
densityMultiplier	float	Optional multiplier 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 13: Renderings of an index-matched medium using different density multipliers (Listing 27)

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 6.7 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 = `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="densityMultiplier" 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 27: A simple heterogeneous medium backed by a grid volume

6.5.2. Homogeneous participating medium (homogeneous)

Parameter	Type	Description
material	string	Name of a material preset, see Table 3. (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)
densityMultiplier	float	Optional multiplier that will be applied to the sigma* parameters. 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 several ways of configuring the medium properties. Either, a material preset can be loaded using the material parameter—see Table 3 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 densityMultiplier parameter can be used to correct the units. For instance, when the scene is in meters and the coefficients are in inverse millimeters, set densityMultiplier 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 28: 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 single channel) and then merge them back into a RGB image. There is a `mtsutil` (Section 4.4) plugin named `joinrgb` that will perform this RGB merging process.

Name	Name
`apple`	`potato`
`chicken1`	`skimmilk`
`chicken2`	`skin1`
`cream`	`skin2`
`ketchup`	`spectralon`
`marble`	`wholemilk`

Table 3: This table lists all supported medium material presets. The values are from Jensen et al. [10] using units of $\frac{1}{mm}$, so remember to set `densityMultiplier` appropriately when your scene is not in units of millimeters. These material names can be used with the plugins `homogeneous`, `dipole`, `hk`, and `sssbrdf`.

6.6. 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`).

6.6.1. Isotropic phase function (`isotropic`)

(a) Isotropic (b) Anisotropic micro-flakes

Figure 14: 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.

6.6.2. 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 [7]. It is parameterizable from backward- ($g < 0$) through isotropic- ($g = 0$) to forward ($g > 0$) scattering.

6.6.3. 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.

6.6.4. Kajiya-Kay phase function (kkay)

This plugin implements the Kajiya-Kay [11] phase function for volumetric rendering of fibers, e.g. hair or cloth.

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

6.6.5. 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 [9].

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 [25].

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.

6.6.6. 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, 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.

6.7. 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.

6.7.1. 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.

6.7.2. 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', 'O', 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 Woocock-Tracking integration method in `heterogeneous` will produce incorrect results.

6.7.3. 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.

```xml
<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 29: Definition of a heterogeneous medium with homogeneous contents

6.8. Luminaires

TBD

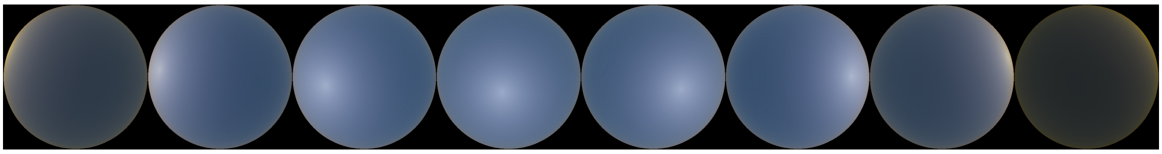
6.8.1. Environment map luminaire (`envmap`)

Parameter	Type	Description
`intensityScale`	`float`	This parameter can be used to scale the the amount of illumination emitted by the luminaire. (Default: 1)

This plugin implements a simple environment map luminaire with importance sampling. It uses the scene's bounding sphere to simulate an infinitely far-away light source and expects an EXR image in latitude-longitude (equirectangular) format.

6.8.2. Skylight luminaire (`sky`)

Parameter	Type	Description
`turbidity`	`float`	This parameter determines the amount of scattering particles (or 'haze') in the atmosphere. Smaller values (~ 2) produce a clear blue sky, larger values (~ 8) lead to an overcast sky, and a very high values (~ 20) cause a color shift towards orange and red. (Default: 3)
`day`	`integer`	Solar day used to compute the sun's position. Must be in the range between 1 and 365. (Default: 180)
`time`	`float`	Fractional time used to compute the sun's position. A time of 4:15 PM corresponds to 16.25. (Default: 15.00)
`latitude`, `longitude`	`float`	These two parameters specify the oberver's latitude and longitude in degrees, which are required to compute the sun's position. (Default: 35.6894, 139.6917 — Tokyo, Japan)
`standardMeridian`	`integer`	Denotes the standard meridian of the time zone for finding the sun's position (Default: 135 — Japan standard time)
`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 (`day`, `time`, `latitude`, `longitude`, and `standardMeridian`) are unnecessary. (Default: none)
`extend`	`boolean`	Extend luminaire below the horizon? (Default: `false`)
`resolution`	`integer`	Specifies the resolution of the precomputed image that is used to represent the sky environment map (Default: 256)
`skyScale`	`float`	This parameter can be used to scale the the amount of illumination emitted by the sky luminaire, for instance to change its units. To switch from photometric ($W/m^2 \cdot sr$) to arbitrary but convenient units in the $[0,1]$ range, set this parameter to `1e-5`. (Default: 1)

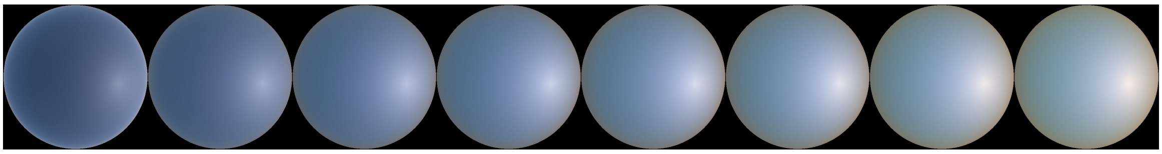


(a) 6AM (b) 8AM (c) 10AM (d) 12PM (e) 2PM (f) 4PM (g) 6PM (h) 8PM

Figure 15: Time series with the default settings (shown by projecting the sky onto a disk. East is left.)

This plugin implements the physically-based skylight model proposed by Preetham et al. [17]. It can be used for realistic daylight renderings of scenes under clear and overcast skies, assuming that the sky is observed from a position either on or close to the surface of the earth.

Numerous parameters allow changing the both the position on Earth, as well as the time of observation. These are used to compute the sun direction which, together with `turbidity`, constitutes the main parameter of the model. If desired, the sun direction can also be specified manually.



(a) 2 (b) 3 (c) 4 (d) 5 (e) 6 (f) 7 (g) 8 (h) 9

Figure 16: Sky light for different turbidity values (fixed time & location)

Turbidity, the other important parameter, specifies the amount of atmospheric extinction due to larger particles (t_l), as opposed to molecules (t_m). Lower values correspond to a clear sky, and higher values produce illumination resembling that of a hazy, overcast sky. Formally, the turbidity is defined as the ratio between the combined extinction cross-section and the cross-section only due to molecules, i.e. $T = \frac{t_m + t_l}{t_m}$. Values between 1 and 30 are possible, though the model will be most accurate for values between 2 and 6, to which it was fit using numerical optimization.

The default coordinate system of the luminaire 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.

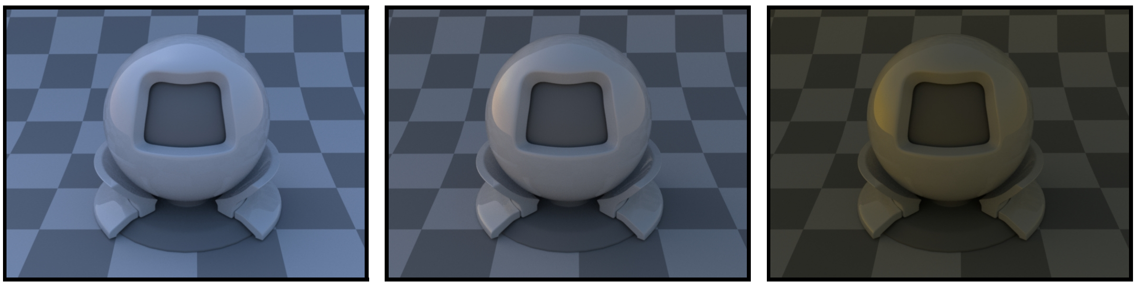
By default, the luminaire will not emit any light below the horizon, which means that these regions will be black when they are observed directly. By setting the `extend` parameter to `true`, the emitted radiance at the horizon will be extended to the entire bottom hemisphere. Note that this will significantly increase the amount of illumination present in the scene.

For performance reasons, the implementation precomputes an environment map of the entire sky that is then forwarded to the `envmap` plugin. The resolution of this environment map can affect the quality of the result. Due to the smoothness of the sky illumination, `resolution` values of around 256 (the default) are usually more than sufficient.

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. The model also currently does not handle cloudy skies. The implementation in Mitsuba is based on code by Preetham et al. It was ported by Tom Kazimiers.

```xml
<luminaire type="sky">
    <transform name="toWorld">
        <rotate x="1" angle="90"/>
    </transform>
</luminaire>
```

Listing 30: Rotating the sky luminaire for scenes that use Z as the "up" direction



(a) 3 PM (b) 6 PM (c) 8 PM

Figure 17: Renderings with the `plastic` material under default conditions

6.8.3. Sun luminaire (sun)

Parameter	Type	Description
turbidity	float	This parameter determines the amount of scattering particles (or 'haze') in the atmosphere. Smaller values (~ 2) produce a clear blue sky, larger values (~ 8) lead to an overcast sky, and a very high values (~ 20) cause a color shift towards orange and red. (Default: 3)
day	integer	Solar day used to compute the sun's position. Must be in the range between 1 and 365. (Default: 180)
time	float	Fractional time used to compute the sun's position. A time of 4:15 PM corresponds to 16.25. (Default: 15.00)
latitude, longitude	float	These two parameters specify the oberver's latitude and longitude in degrees, which are required to compute the sun's position. (Default: 35.6894, 139.6917 — Tokyo, Japan)
standardMeridian	integer	Denotes the standard meridian of the time zone for finding the sun's position (Default: 135 — Japan standard time)
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 (day, time, latitude, longitude, and standardMeridian) are unnecessary. (Default: none)
resolution	integer	Specifies the resolution of the precomputed image that is used to represent the sun environment map (Default: 256)
sunScale	float	This parameter can be used to scale the the amount of illumination emitted by the sun luminaire, for instance to change its units. To switch from photometric ($W/m^2 \cdot sr$) to arbitrary but convenient units in the $[0,1]$ range, set this parameter to 1e-5. (Default: 1)

6.8.4. Sun and sky luminaire (sunsky)

Parameter	Type	Description
turbidity	float	This parameter determines the amount of scattering particles (or 'haze') in the atmosphere. Smaller values (~ 2) produce a clear blue sky, larger values (~ 8) lead to an overcast sky, and a very high values (~ 20) cause a color shift towards orange and red. (Default: 3)
day	integer	Solar day used to compute the sun's position. Must be in the range between 1 and 365. (Default: 180)
time	float	Fractional time used to compute the sun's position. A time of 4:15 PM corresponds to 16.25. (Default: 15.00)
latitude, longitude	float	These two parameters specify the oberver's latitude and longitude in degrees, which are required to compute the sun's position. (Default: 35.6894, 139.6917 — Tokyo, Japan)
standardMeridian	integer	Denotes the standard meridian of the time zone for finding the sun's position (Default: 135 — Japan standard time)
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 (day, time, latitude, longitude, and standardMeridian) are unnecessary. (Default: none)
extend	boolean	Extend luminaire below the horizon? (Default: false)
resolution	integer	Specifies the resolution of the precomputed image that is used to represent the sky environment map (Default: 256)
skyScale	float	This parameter can be used to scale the the amount of illumination emitted by the sky.
sunScale	float	This parameter can be used to scale the the amount of illumination emitted by the sun.

This plugin implements the physically-based skylight model proposed by Preetham et al. [17]. It can be used for realistic daylight renderings of scenes under clear and overcast skies, assuming that the sky is observed from a position either on or close to the surface of the earth.

This is a convenience plugin, which has the sole purpose of instantiating sun and sky at the same time. Please refer to these plugins individually for more detail

6.9. Integrators

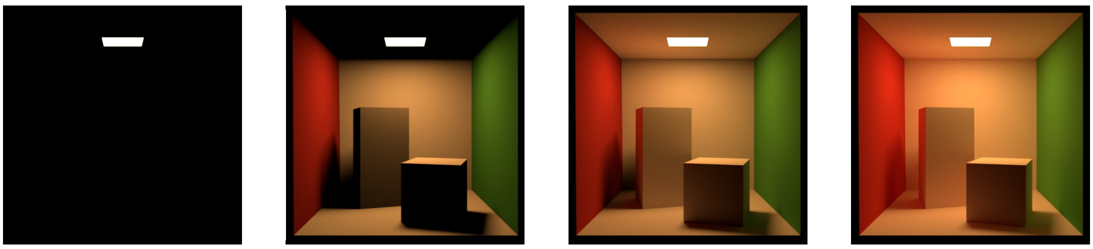
In Mitsuba, the different rendering techniques are collectively referred to as *integrators*, since they perform integration over a high-dimensional space. Each integrator represents a specific approach for solving the light transport equation—usually favored in certain scenarios, but at the same time 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="0.3.1">
    <!-- 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.

Path length



(a) Max. length = 1 (b) Max. length = 2 (c) Max. length = 3 (d) Max. length = ∞

Figure 18: These Cornell box renderings demonstrate the visual effect of a maximum path length. 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 length*. Here, a path refers to a chain of scattering events that starts at the light source and ends at the eye or camera. It is often useful to limit the path length (Figure 18) 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 length unlimited.

Mitsuba counts lengths 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 camera without any scattering interaction in between). A length-2 path (also known as "direct illumination") includes a single scattering event (Figure 19).

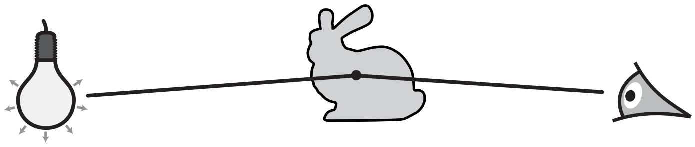


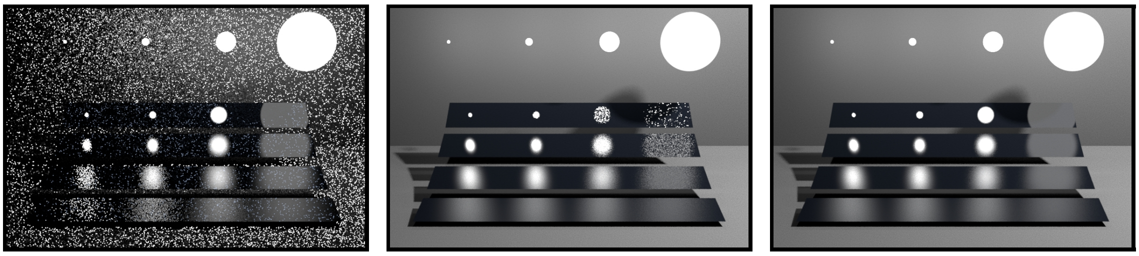
Figure 19: A ray of emitted light is scattered by an object and subsequently reaches the eye/camera. In Mitsuba, this is a *length-2* path, since it has two edges.

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).

6.9.1. Direct illumination integrator (`direct`)

Parameter	Type	Description
`shadingSamples`	`integer`	This convenience parameter can be used to set both `luminaireSamples` and `bsdfSamples` at the same time.
`luminaireSamples`	`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 luminaires (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`)



(a) Only BSDF sampling **(b)** Only luminaire sampling **(c)** BSDF and luminaire sampling

Figure 20: 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 luminaire 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 20a), while the opposite is true for the luminaire sampling technique (Figure 20b). By combining these approaches, one can obtain a rendering technique that works well in both cases (Figure 20c).

The number of samples spent on either technique is configurable, hence it is also possible to turn this plugin into an luminaire 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.

6.9.2. 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: `10`)
`strictNormals`	`boolean`	Be strict about potential inconsistencies involving shading normals? See `path` for details. (Default: no, i.e. `false`)

This integrator implements a path tracer that makes use of *multiple importance sampling*: for each surface interaction, the integrator generates a single BSDF and emitter sample and combines them using the power heuristic.

For best results, combine the path tracer with the low-discrepancy sample generator (`ldsampler`).

Remarks:

- This integrator does not handle participating media

6.10. Films

This section contains a reference of the film plugins that come with Mitsuba. A film defines how conducted measurements are stored and coverted into a final output format.

6.10.1. MATLAB M-file film (`mfilm`)

Parameter	Type	Description
`width, height`	`integer`	Width and height of the camera sensor in pixels (Default: 768, 576)
`cropOffsetX, cropOffsetY, cropWidth, cropHeight`	`integer`	These parameter can optionally be provided to render a sub-rectangle of the output (Default: Unused)
`alpha`	`boolean`	Include an alpha channel in the output image? (Default: `true`)
`banner`	`boolean`	Include a small Mitsuba banner in the output image? (Default: `true`)

This plugin provides a camera film that exports luminance values as a matrix using the MATLAB M-file format. 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 a test suite.

When Mitsuba is started with the "test case mode" parameter (`-t`), this class will write triples consisting of the luminance, variance, and sample count for every pixel (instead of just the luminance).

6.10.2. PNG-based film (`pngfilm`)

Parameter	Type	Description
`width, height`	`integer`	Width and height of the camera sensor in pixels (Default: 768, 576)
`cropOffsetX, cropOffsetY, cropWidth, cropHeight`	`integer`	These parameter can optionally be provided to render a sub-rectangle of the output (Default: Unused)
`alpha`	`boolean`	Include an alpha channel in the output image? (Default: `true`)
`banner`	`boolean`	Include a small Mitsuba banner in the output image? (Default: `true`)
`toneMappingMethod`	`string`	Specifies the tonemapping method that should be used to convert high-dynamic range images to 8 bits per channel. 1. `gamma`: Use a basic Gamma conversion 2. `reinhard`: Use a global version of the Reinhard [18] tonemapping method
`reinhardKey, reinhardBurn`	`float`	When `toneMappingMethod=reinhard`, these two parameters specify the *key* and *burn* parameters of that model. (Default: `reinhardKey`=0.18 and `reinhardBurn`=0)

This plugin implements a simple camera film that stores the captured image as an RGBA-based low dynamic-range PNG file with gamma correction. The measured spectral power distributions are converted to linear RGB based on CIE 1931 XYZ color matching functions and ITU-R Rec. BT.709. If desired, the class can optionally apply a global version of the Reinhard tonemapping algorithm.

6.10.3. OpenEXR-based film (`exrfilm`)

Parameter	Type	Description
`width, height`	`integer`	Width and height of the camera sensor in pixels (Default: 768, 576)
`cropOffsetX, cropOffsetY, cropWidth, cropHeight`	`integer`	These parameter can optionally be provided to render a sub-rectangle of the output (Default: Unused)
`alpha`	`boolean`	Include an alpha channel in the output image? (Default: `true`)
`banner`	`boolean`	Include a small Mitsuba banner in the output image? (Default: `true`)

This plugin implements a simple camera film that stores the captured image as an RGBA-based high dynamic-range EXR file. It does not perform any gamma correction (i.e. the EXR file will contain linear radiance values).

The measured spectral power distributions are converted to linear RGB based on CIE 1931 XYZ color matching functions and ITU-R Rec. BT.709.

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.

7. 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.

8. 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. luminaires) don't actually provide a header file, hence they can only be accessed using the generic `Luminaire` 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.

9. 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 `SampleIntegrator` 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.

9.1. 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 SampleIntegrator {
public:
    MTS_DECLARE_CLASS()
};

MTS_IMPLEMENT_CLASS_S(MyIntegrator, false, SampleIntegrator)
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 (`SampleIntegrator`).

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) : SampleIntegrator(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)
     : SampleIntegrator(stream, manager) {
        m_color = Spectrum(stream);
    }

    /// Serialize to a binary data stream
    void serialize(Stream *stream, InstanceManager *manager) const {
        SampleIntegrator::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.

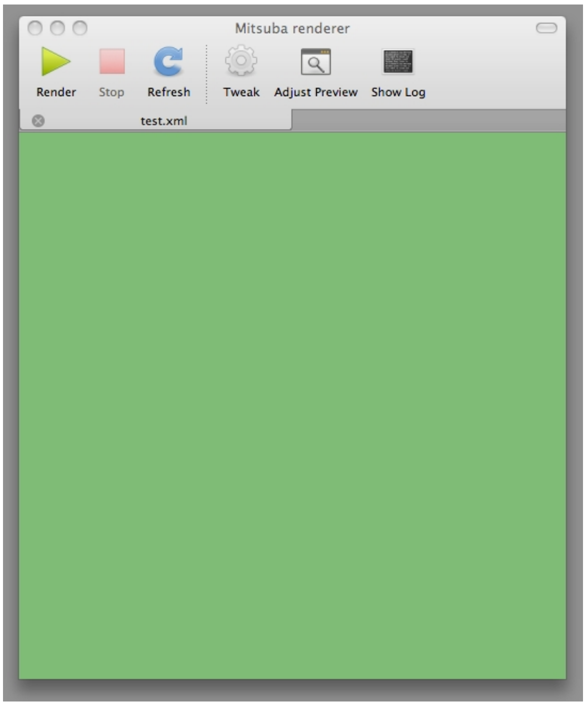
The last thing to add is a function, which returns an estimate for the radiance along a ray differential: here, we simply return the stored color

```
    /// 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 `SConstruct` file in the main directory, and add the following line after the comment "`# Integrators`":

```
plugins += env.SharedLibrary('plugins/myIntegrator', ['src/integrators/
    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.

9.2. 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) {
    SampleIntegrator::preprocess(scene, queue, job, sceneResID,
        cameraResID, samplerResID);

    const AABB &sceneAABB = scene->getAABB();
    Point cameraPosition = scene->getCamera()->getPosition();
```

```
        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;
    }
```

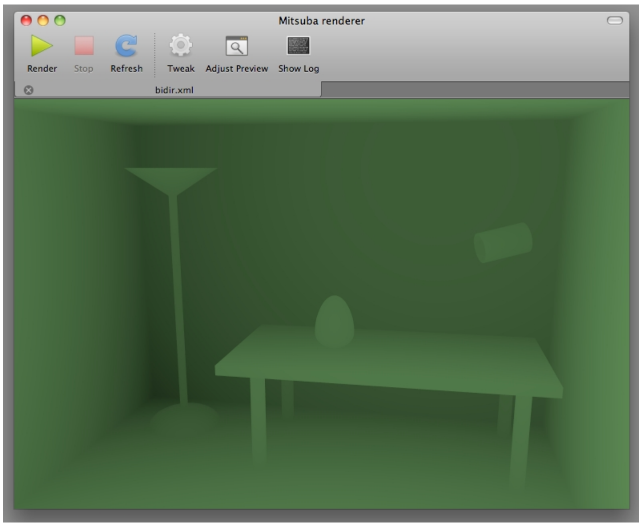
The bottom of this function should be relatively self-explanatory. The numerous arguments at the top are related to the parallelization layer, which will be considered in more detail in the next section. Briefly, the render queue provides synchronization facilities for render jobs (e.g. one can wait 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*.

9.3. 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="errctrl">
    <!-- 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 `SampleIntegrator::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;
   }
```

10. 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('plugins/rot13', ['src/utils/rot13.cpp'])
```

into the SConscript (near the comment "`Build the plugins - utilities`"). 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" ..
```

 10. Parallelization layer

```
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
```

11. Python integration

A recent feature of Mitsuba is a simple Python interface to the renderer API. While the interface is still limited at this point, it can already be used for many useful purposes. To access the API, start your Python interpreter and enter

```
import mitsuba
```

For this to work on MacOS X, you will first have to run the "*Apple Menu→Command-line access*" menu item from within Mitsuba. On Windows and non-packaged Linux builds, you may have to update the extension search path before issuing the `import` command, e.g.:

```
import sys

# Update the extension search path
# (may vary depending on your setup)
sys.path.append('dist/python')

import mitsuba
```

For an overview of the currently exposed API subset, please refer to the following page: http://www.mitsuba-renderer.org/api/group__libpython.html.

11.1. 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, overridable virtual function calls, 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)
```

11.2. Recipes

The following section contains a series of "recipes" on how to do certain things with the help of the Python bindings.

11.2.1. 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()

# Add the search path needed to load plugins
fileResolver.addPath('<path to mitsuba directory>')

# Add the search path needed to load scene resources
fileResolver.addPath('<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)
```

11.2.2. 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('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())
```

11.2.3. 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 4.3). 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()
```

11.2.4. 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 camera with a film that writes PNG images:

```
from mitsuba.core import *
pmgr = PluginManager.getInstance()

# Encodes parameters on how to instantiate the 'perspective' plugin
cameraProps = Properties('perspective')
cameraProps['toWorld'] = Transform.lookAt(
    Point(0, 0, -10),  # Camera origin
    Point(0, 0, 0),    # Camera target
    Vector(0, 1, 0)    # 'up' vector
)
cameraProps['fov'] = 45.0

# Encodes parameters on how to instantiate the 'pngfilm' plugin
filmProps = Properties('pngfilm')
filmProps['width'] = 1920
filmProps['height'] = 1080

# Load and instantiate the plugins
camera = pmgr.createObject(cameraProps)
```

```
film = pmgr.createObject(filmProps)

# First configure the film and then add it to the camera
film.configure()
camera.addChild('film', film)

# Now, the camera can be configured
camera.configure()
```

The above code fragment uses the plugin manager to construct a `Camera` 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 `pngfilm.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()
camera = pmgr.create({
    'type' : 'perspective',
    'toWorld' : Transform.lookAt(
        Point(0, 0, -10),
        Point(0, 0, 0),
        Vector(0, 1, 0)
    ),
    'film' : {
        'type' : 'pngfilm',
        '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 camera, 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' : 'pngfilm',
        '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()
```

11.2.5. 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.

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')
```

12. Acknowledgments

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. The architecture of the coherent path tracer traversal code was influenced by Radius from Thierry Berger-Perrin. Many thanks go to my advisor Steve Marschner, who let me spend time on this project. 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. [15] 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
- Tom Kazimiers
- Marios Papas
- Edgar Velázquez-Armendáriz
- Jirka Vorba

Mitsuba makes heavy use of the following amazing libraries and tools:

- Qt 4 by Nokia
- OpenEXR by Industrial Light & Magic
- Xerces-C++ by the Apache Foundation
- Eigen by Benoît Jacob and Gaël Guennebaud
- 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 by the Independent JPEG Group
- libpng by Guy Eric Schalnat, Andreas Dilger, Glenn Randers-Pehrson and others
- libply by Ares Lagae
- BWToolkit by Brandon Walkin
- POSIX Threads for Win32 by Ross Johnson
- The SCons build system by the SCons Foundation

References

[1] Ashikhmin, M., and Shirley, P. An anisotropic phong BRDF model. *Graphics tools: The jgt editors' choice* (2005), 303.

[2] 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.

[3] 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.

[4] Dür, A. An Improved Normalization For The Ward Reflectance Model. *Journal of graphics, gpu, and game tools 11*, 1 (2006), 51–59.

[5] 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.

[6] 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.

[7] Henyey, L., and Greenstein, J. Diffuse radiation in the galaxy. *The Astrophysical Journal 93* (1941), 70–83.

[8] Irawan, P. *Appearance of woven cloth*. PhD thesis, Cornell University, Ithaca, NY, USA, 2008. http://ecommons.library.cornell.edu/handle/1813/8331.

[9] 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.

[10] Jensen, H., Marschner, S., 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* (2001), ACM, pp. 511–518.

[11] Kajiya, J., and Kay, T. Rendering fur with three dimensional textures. *ACM Transactions on Graphics 23*, 3 (1989), 271–280.

[12] Lafortune, E. P., and Willems, Y. D. Using the modified phong reflectance model for physically based rendering. Tech. rep., Cornell University, 1994.

[13] 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.

[14] 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.

[15] Palik, E., and Ghosh, G. *Handbook of optical constants of solids*. Academic press, 1998.

[16] Phong, B.-T. Illumination for Computer Generated Pictures. *Communications of the ACM 18*, 6 (1975), 311–317.

[17] 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.

[18] Reinhard, E., Stark, M., Shirley, P., and Ferwerda, J. Photographic tone reproduction for digital images. *ACM Transactions on Graphics 21*, 3 (2002), 267–276.

[19] 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.

[20] Smits, B. An RGB-to-spectrum conversion for reflectances. *Graphics tools: The jgt editors' choice* (2005), 291.

[21] Walter, B. Notes on the ward brdf. Tech. Rep. PCG-05-06, Program of Computer Graphics, Cornell University, 2005.

[22] 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).

[23] 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.

[24] 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.

[25] 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.