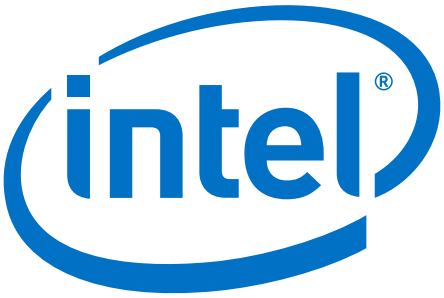

INTEL® OPEN IMAGE DENOISE

HIGH-PERFORMANCE DENOISING LIBRARY FOR RAY TRACING

Version 0.8.1
February 11, 2019

Contents

Chapter 1
Open Image Denoise Overview

Intel® Open Image Denoise is a collection of high-performance, high-quality denoising filters for images rendered with ray tracing. Open Image Denoise is part of the Intel Rendering Framework and is released under the permissive Apache 2.0 license.

The purpose of Open Image Denoise is to provide an open, high-quality, efficient, and easy-to-use denoising library that allows one to significantly reduce rendering times in ray tracing based rendering applications. It filters out the Monte Carlo noise inherent to stochastic ray tracing methods like path tracing, reducing the amount of necessary samples per pixel by even multiple orders of magnitude (depending on the desired closeness to the ground truth). A simple but flexible C/C++ API ensures that the library can be easily integrated into most existing or new rendering solutions.

At the heart of the Open Image Denoise library is an efficient deep learning based denoising filter, which was trained to handle a wide range of samples per pixel (spp), from 1 spp to almost fully converged. Thus it is suitable for both preview and final-frame rendering. The filters can denoise images either using only the noisy color (*beauty*) buffer, or, to preserve as much detail as possible, can optionally utilize auxiliary feature buffers as well (e.g. albedo, normal). Such buffers are supported by most renderers as arbitrary output variables (AOVs) or can be usually implemented with little effort.

Open Image Denoise supports Intel® 64 architecture based CPUs and compatible architectures, and runs on anything from laptops, to workstations, to compute nodes in HPC systems. It is efficient enough to be suitable not only for offline rendering, but, depending on the hardware used, also for interactive ray tracing.

Open Image Denoise internally builds on top of Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN), and automatically exploits modern instruction sets like Intel SSE4, AVX2, and AVX-512 to achieve high denoising performance. A CPU with support for at least SSE4.1 is required to run Open Image Denoise.

Support and Contact

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

For recent news, updates, and announcements, please see our complete news/updates page.

Join our mailing list to receive release announcements and major news regarding Open Image Denoise.

Version History

Changes in v0.8.1:

- Fixed wrong path to TBB in the generated CMake configs
- Fixed wrong rpath in the binaries
- Fixed compile error on some macOS systems
- Fixed minor compile issues with Visual Studio
- Lowered the CPU requirement to SSE4.1
- Minor example update

Changes in v0.8.0:

- Initial beta release

Chapter 2

Building Open Image Denoise from Source

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

Prerequisites

Open Image Denoise currently supports 64-bit Linux, Windows, and macOS operating systems. In addition, before you can build Open Image Denoise you need the following prerequisites:

- You can clone the latest Open Image Denoise sources via:

  ```
  git clone --recursive https://github.com/OpenImageDenoise/oidn.git
  ```

- To build Open Image Denoise you need CMake 3.1 or later, a C++11 compiler (we recommend using Clang, but also support GCC, Microsoft Visual Studio 2015 or later, and Intel® C++ Compiler 17.0 or later), and Python 2.7 or later.

- Additionally you require a copy of Intel® Threading Building Blocks (TBB) 2017 or later.

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

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

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

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

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

Under macOS these dependencies can be installed using MacPorts:

```
sudo port install cmake tbb
```

Under Windows please directly use the appropriate installers or packages for CMake, Python, and TBB.

Compiling Open Image Denoise on Linux/macOS

Assuming the above prerequisites are all fulfilled, building Open Image Denoise through CMake is easy:

- Create a build directory, and go into it

  ```
  mkdir oidn/build
  cd oidn/build
  ```

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

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

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

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

- Open the CMake configuration dialog

  ```
  ccmake ..
  ```

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

  ```
  make
  ```

- You should now have `libOpenImageDenoise.so` as well as a set of example applications.

Compiling Open Image Denoise on Windows

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

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

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

- If the configuration fails because some dependencies could not be found then follow the instructions given in the error message, e.g., set the variable `TBB_ROOT` to the folder where TBB was installed.

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

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

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

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

Use `-D` to set variables for CMake, e.g., the path to TBB with "`-D TBB_ROOT=\path\to\tbb`".

Chapter 3
Open Image Denoise API

Open Image Denoise provides a C99 API (also compatible with C++) and a C++11 wrapper API as well. For simplicity, this document mostly refers to the C99 version of the API.

The API is designed in an object-oriented manner, e.g. it contains device objects (`OIDNDevice` type), buffer objects (`OIDNBuffer` type), and filter objects (`OIDNFilter` type). All objects are reference-counted, and handles can be released by calling the appropriate release function (e.g. `oidnReleaseDevice`) or retained by incrementing the reference count (e.g. `oidnRetainDevice`).

An important aspect of objects is that setting their parameters do not have an immediate effect (with a few exceptions). Instead, objects with updated parameters are in an unusable state until the parameters get explicitly committed to a given object. The commit semantic allows for batching up multiple small changes, and specifies exactly when changes to objects will occur.

All API calls are thread-safe, but operations that use the same device will be serialized, so the amount of API calls from different threads should be minimized.

To have a quick overview of the C99 and C++11 APIs, see the following simple example code snippets.

C99 API Example

```
#include <OpenImageDenoise/oidn.h>
...
// Create an Open Image Denoise device
OIDNDevice device = oidnNewDevice(OIDN_DEVICE_TYPE_DEFAULT);
oidnCommitDevice(device);

// Create a denoising filter
OIDNFilter filter = oidnNewFilter(device, "RT"); // generic ray tracing filter
oidnSetSharedFilterImage(filter, "color",  colorPtr,
                         OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0);
oidnSetSharedFilterImage(filter, "albedo", albedoPtr,
                         OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // optional
oidnSetSharedFilterImage(filter, "normal", normalPtr,
                         OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // optional
oidnSetSharedFilterImage(filter, "output", outputPtr,
                         OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0);
oidnSetFilter1b(filter, "hdr", true); // image is HDR
oidnCommitFilter(filter);

// Filter the image
oidnExecuteFilter(filter);
```

```
// Check for errors
const char* errorMessage;
if (oidnGetDeviceError(device, &errorMessage) != OIDN_ERROR_NONE)
  printf("Error: %s\n", errorMessage);

// Cleanup
oidnReleaseFilter(filter);
oidnReleaseDevice(device);
```

C++11 API Example

```
#include <OpenImageDenoise/oidn.hpp>
...
// Create an Open Image Denoise device
oidn::DeviceRef device = oidn::newDevice();
device.commit();

// Create a denoising filter
oidn::FilterRef filter = device.newFilter("RT"); // generic ray tracing filter
filter.setImage("color",  colorPtr,  oidn::Format::Float3, width, height);
filter.setImage("albedo", albedoPtr, oidn::Format::Float3, width, height); // optional
filter.setImage("normal", normalPtr, oidn::Format::Float3, width, height); // optional
filter.setImage("output", outputPtr, oidn::Format::Float3, width, height);
filter.set("hdr", true); // image is HDR
filter.commit();

// Filter the image
filter.execute();

// Check for errors
const char* errorMessage;
if (device.getError(errorMessage) != oidn::Error::None)
  std::cout << "Error: " << errorMessage << std::endl;
```

Device

Open Image Denoise supports a device concept, which allows different components of the application to use the Open Image Denoise API without interfering with each other. An application first needs to create a device with

```
OIDNDevice oidnNewDevice(OIDNDeviceType type);
```

where the `type` enumeration maps to a specific device implementation, which can be one of the following:

Name	Description
OIDN_DEVICE_TYPE_DEFAULT	select the approximately fastest device
OIDN_DEVICE_TYPE_CPU	CPU device (requires SSE4.1 support)

Table 3.1 – Supported device types, i.e., valid constants of type `OIDNDeviceType`.

Once a device is created, you can call

```
void oidnSetDevice1b(OIDNDevice device, const char* name, bool value);
void oidnSetDevice1i(OIDNDevice device, const char* name, int  value);
bool oidnGetDevice1b(OIDNDevice device, const char* name);
int  oidnGetDevice1i(OIDNDevice device, const char* name);
```

to set and get parameter values on the device. Note that some parameters are constants, thus trying to set them is an error. See the tables below for the parameters supported by devices.

Type	Name	Description
const int	version	combined version number (major.minor.patch) with two decimal digits per component
const int	versionMajor	major version number
const int	versionMinor	minor version number
const int	versionPatch	patch version number

Table 3.2 – Parameters supported by all devices.

Type	Name	Default	Description
int	numThreads	0	maximum number of threads which Open Image Denoise should use; 0 will set it automatically to get the best performance
bool	setAffinity	true	bind software threads to hardware threads if set to true (improves performance); false disables binding

Table 3.3 – Additional parameters supported only by CPU devices.

Note that the CPU device heavily relies on setting the thread affinities to achieve optimal performance, so it is highly recommended to leave this option enabled. However, this may interfere with the application if that also sets the thread affinities, potentially causing performance degradation. In such cases, the recommended solution is to either disable setting the affinities in the application or in Open Image Denoise, or to always set/reset the affinities before/after each parallel region in the application (e.g., if using TBB, with `tbb::task_arena` and `tbb::task_scheduler_observer`).

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

```
void oidnCommitDevice(OIDNDevice device);
```

This device can then be used to construct further objects, such as buffers and filters. Note that a device can be committed only once during its lifetime. Before the application exits, it should release all devices by invoking

```
void oidnReleaseDevice(OIDNDevice device);
```

Note that Open Image Denoise uses reference counting for all object types, so this function decreases the reference count of the device, and if the count reaches 0 the device will automatically get deleted. It is also possible to increase the reference count by calling

```
void oidnRetainDevice(OIDNDevice device);
```

An application typically creates only a single device. If required differently, it should only use a small number of devices at any given time.

Error Handling

Each user thread has its own error code per device. If an error occurs when calling an API function, this error code is set to the occurred error if it stores no previous error. The currently stored error can be queried by the application via

```
OIDNError oidnGetDeviceError(OIDNDevice device, const char** outMessage);
```

where `outMessage` can be a pointer to a C string which will be set to a more descriptive error message, or it can be `NULL`. This function also clears the error code, which assures that the returned error code is always the first error occurred since the last invocation of `oidnGetDeviceError` on the current thread. Note that the optionally returned error message string is valid only until the next invocation of the function.

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

```
typedef void (*OIDNErrorFunction)(void* userPtr, OIDNError code, const char* message);
```

via

```
void oidnSetDeviceErrorFunction(OIDNDevice device, OIDNErrorFunction func, void* userPtr);
```

to get notified when errors occur. Only a single callback function can be registered per device, and further invocations overwrite the previously set callback function, which do *not* require also calling the `oidnCommitDevice` function. Passing `NULL` as function pointer disables the registered callback function. When the registered callback function is invoked, it gets passed the user-defined payload (`userPtr` argument as specified at registration time), the error code (`code` argument) of the occurred error, as well as a string (`message` argument) that further describes the error. The error code is always set even if an error callback function is registered. It is recommended to always set a error callback function, to detect all errors.

When the device construction fails, `oidnNewDevice` returns `NULL` as device. To detect the error code of a such failed device construction, pass `NULL` as device to the `oidnGetDeviceError` function. For all other invocations of `oidnGetDeviceError`, a proper device handle must be specified.

The following errors are currently used by Open Image Denoise:

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

Name	Description
OIDN_ERROR_NONE	no error occurred
OIDN_ERROR_UNKNOWN	an unknown error occurred
OIDN_ERROR_INVALID_ARGUMENT	an invalid argument was specified
OIDN_ERROR_INVALID_OPERATION	the operation is not allowed
OIDN_ERROR_OUT_OF_MEMORY	not enough memory to execute the operation
OIDN_ERROR_UNSUPPORTED_HARDWARE	the hardware (e.g., CPU) is not supported

Buffer

Large data like images can be passed to Open Image Denoise either via pointers to memory allocated and managed by the user (this is the recommended, often easier and more efficient approach, if supported by the device) or by creating buffer objects (supported by all devices). To create a new data buffer with memory allocated and owned by the device, holding `byteSize` number of bytes, use

```
OIDNBuffer oidnNewBuffer(OIDNDevice device, size_t byteSize);
```

The created buffer is bound to the specified device (`device` argument). The specified number of bytes are allocated at buffer construction time and deallocated when the buffer is destroyed.

It is also possible to create a "shared" data buffer with memory allocated and managed by the user with

```
OIDNBuffer oidnNewSharedBuffer(OIDNDevice device, void* ptr, size_t byteSize);
```

where `ptr` points to the user-managed memory and `byteSize` is its size in bytes. At buffer construction time no buffer data is allocated, but the buffer data provided by the user is used. The buffer data must remain valid for as long as the buffer may be used, and the user is responsible to free the buffer data when no longer required.

Similar to device objects, buffer objects are also reference-counted and can be retained and released by calling the following functions:

```
void oidnRetainBuffer(OIDNBuffer buffer);
void oidnReleaseBuffer(OIDNBuffer buffer);
```

Accessing the data stored in a buffer object is possible by mapping it into the address space of the application using

```
void* oidnMapBuffer(OIDNBuffer buffer, OIDNAccess access, size_t byteOffset, size_t byteSize)
```

where `access` is the desired access mode of the mapped memory, `byteOffset` is the offset to the beginning of the mapped memory region in bytes, and `byteSize` is the number of bytes to map. The function returns a pointer to the mapped buffer data. If the specified `byteSize` is `0`, the maximum available amount of memory will be mapped. The `access` argument must be one of the access modes in the following table:

Name	Description
OIDN_ACCESS_READ	read-only access
OIDN_ACCESS_WRITE	write-only access
OIDN_ACCESS_READ_WRITE	read and write access
OIDN_ACCESS_WRITE_DISCARD	write-only access but the previous contents will be discarded

Table 3.5 – Access modes for memory regions mapped with `oidnMapBuffer`, i.e., valid constants of type `OIDNAccess`.

After accessing the mapped data in the buffer, the memory region must be unmapped with

```
void oidnUnmapBuffer(OIDNBuffer buffer, void* mappedPtr);
```

where `mappedPtr` must be a pointer returned by a call to `oidnMapBuffer` for the specified buffer. Any change to the mapped data is guaranteed to take effect only after unmapping the memory region.

Data Format

Buffers store opaque data and thus have no information about the type and format of the data. Other objects, e.g. filters, typically require specifying the format of the data stored in buffers or shared via pointers. This can be done using the `OIDNFormat` enumeration type:

Name	Description
OIDN_FORMAT_UNDEFINED	undefined format
OIDN_FORMAT_FLOAT	32-bit single-precision floating point scalar
OIDN_FORMAT_FLOAT[234]	... and [234]-element vector

Table 3.6 – Supported data formats, i.e., valid constants of type `OIDNFormat`.

Filter

Filters are the main objects in Open Image Denoise that are responsible for the actual denoising. The library ships with a collection of filters which are optimized for different types of images and use cases. To create a filter object, call

```
OIDNFilter oidnNewFilter(OIDNDevice device, const char* type);
```

where `type` is the name of the filter type to create. The supported filter types are documented later in this section. Once created, filter objects can be retained and released with

```
void oidnRetainFilter(OIDNFilter filter);
void oidnReleaseFilter(OIDNFilter filter);
```

After creating a filter, it needs to be set up by specifying the input and output image buffers, and potentially setting other parameter values as well.

To bind image buffers to the filter, you can use one of the following functions:

```
void oidnSetFilterImage(OIDNFilter filter, const char* name,
                        OIDNBuffer buffer, OIDNFormat format,
                        size_t width, size_t height,
                        size_t byteOffset,
                        size_t bytePixelStride, size_t byteRowStride);

void oidnSetSharedFilterImage(OIDNFilter filter, const char* name,
                              void* ptr, OIDNFormat format,
                              size_t width, size_t height,
                              size_t byteOffset,
                              size_t bytePixelStride, size_t byteRowStride);
```

It is possible to specify either a data buffer object (`buffer` argument) with the `oidnSetFilterImage` function, or directly a pointer to shared user-managed data (`ptr` argument) with the `oidnSetSharedFilterImage` function.

In both cases, you must also specify the name of the image parameter to set (`name` argument, e.g. `"color"`, `"output"`), the pixel format (`format` argument), the width and height of the image in number of pixels (`width` and `height` arguments), the starting offset of the image data (`byteOffset` argument), the pixel stride (`bytePixelStride` argument) and the row stride (`byteRowStride` argument), in number of bytes. Note that the row stride must be an integer multiple of the pixel stride.

If the pixels and/or rows are stored contiguously (tightly packed without any gaps), you can set `bytePixelStride` and/or `byteRowStride` to `0` to let the library compute the actual strides automatically, as a convenience.

Filters may have parameters other than buffers as well, which you can set and get using the following functions:

```
void oidnSetFilter1b(OIDNFilter filter, const char* name, bool value);
void oidnSetFilter1i(OIDNFilter filter, const char* name, int  value);
bool oidnGetFilter1b(OIDNFilter filter, const char* name);
int  oidnGetFilter1i(OIDNFilter filter, const char* name);
```

After setting all necessary parameters for the filter, the changes must be commmitted by calling

```
void oidnCommitFilter(OIDNFilter filter);
```

The parameters can be updated after committing the filter, but it must be re-committed for the changes to take effect.

Finally, an image can be filtered by executing the filter with

```
void oidnExecuteFilter(OIDNFilter filter);
```

which will read the input image data from the specified buffers and produce the denoised output image.

In the following we describe the different filters that are currently implemented in Open Image Denoise.

RT

The `RT` (**r**ay **t**racing) filter is a generic ray tracing denoising filter which is suitable for denoising images rendered with Monte Carlo ray tracing methods like unidirectional and bidirectional path tracing. It supports depth of field and motion blur as well, but it is *not* temporally stable. The filter is based on a deep learning based denoising algorithm, and it aims to provide a good balance between denoising performance and quality for a wide range of samples per pixel.

It accepts either a low dynamic range (LDR) or high dynamic range (HDR) color image as input. Optionally, it also accepts auxiliary *feature* images, e.g. albedo and normal, which improve the denoising quality, preserving more details in the image.

The `RT` filter has certain limitations regarding the supported input images. Most notably, it cannot denoise images that were not rendered with ray tracing. Another important limitation is related to anti-aliasing filters. Most renderers use a high-quality pixel reconstruction filter instead of the trivial box filter to minimize aliasing artifacts (e.g. Gaussian, Blackman-Harris). The `RT` filter does support such pixel filters but only if implemented with importance sampling. Weighted pixel sampling (sometimes called *splatting*) introduces correlation between neighboring pixels, which causes the denoising to fail (the noise will not be filtered), thus it is not supported.

The filter can be created by passing `"RT"` to the `oidnNewFilter` function as the filter type. The filter supports the following parameters:

Table 3.7 – Parameters supported by the `RT` filter.

Type	Format	Name	Default	Description
Image	float3	color		input color image (LDR values in $[0, 1]$ or HDR values in $[0, +\infty)$)
Image	float3	albedo		input feature image containing the albedo (values in $[0, 1]$) of the first hit per pixel; *optional*
Image	float3	normal		input feature image containing the shading normal (world-space or view-space, arbitrary length, values in $(-\infty, +\infty)$) of the first hit per pixel; *optional*, requires setting the albedo image too
Image	float3	output		output image; can be one of the input images
bool		hdr	false	whether the color is HDR
bool		srgb	false	whether the color is encoded with the sRGB (2.2 gamma) curve (LDR only) or is linear; the output will be encoded with the same curve

All specified images must have the same dimensions.

Figure 3.1 – Example noisy color image rendered using unidirectional path tracing (512 spp). *Scene by Evermotion.*

Figure 3.2 – Example output image denoised using color and auxiliary (first-hit) feature images (albedo and normal)

Using auxiliary feature images like albedo and normal helps preserving fine details and textures in the image thus can significantly improve denoising quality. These images should typically contain feature values for the first hit (i.e. the surface which is directly visible) per pixel. This works well for most surfaces but does not provide any benefits for reflections and objects visible through transparent surfaces (compared to just using the color as input). However, in certain cases this issue can be fixed by storing feature values for a subsequent hit (i.e. the reflection and/or refraction) instead of the first hit. For example, it usually works well to follow perfect specular (*delta*) paths and store features for the first diffuse or glossy surface hit instead (e.g. for perfect specular dielectrics and mirrors). This can greatly improve the quality of reflections and transmission. We will describe this approach in more detail in the following subsections.

The auxiliary feature images should be as noise-free as possible. It is not a strict requirement but too much noise in the feature images may cause residual noise in the output. Also, all feature images should use the same pixel reconstruction filter as the color image. Using a properly anti-aliased color image but aliased albedo or normal images will likely introduce artifacts around edges.

Albedo

The albedo image is the feature image that usually provides the biggest quality improvement. It should contain the approximate color of the surfaces independent of illumination and viewing angle.

For simple matte surfaces this means using the diffuse color/texture as the albedo. For other, more complex surfaces it is not always obvious what is the best way to compute the albedo, but the denoising filter is flexibile to a certain extent and works well with differently computed albedos. Thus it is not necessary to compute the strict, exact albedo values but must be always between `0` and `1`.

For metallic surfaces the albedo should be either the reflectivity at normal incidence (e.g. from the artist friendly metallic Fresnel model) or the average reflectivity, or if these are constant (not textured) or unknown, the albedo can be simply 1 as well.

The albedo for dielectric surfaces (e.g. glass) should be either 1 or, if the surface is perfect specular (i.e. has a delta BSDF), the Fresnel blend of the reflected and transmitted albedos (as previously discussed). The latter usually works better but *only* if it does not introduce too much additional noise due to random sampling. Thus we recommend to split the path into a reflected and a transmitted path at the first hit, and perhaps fall back to an albedo of 1 for subsequent dielectric hits, to avoid noise. The reflected albedo can be used for mirror-like surfaces as well.

The albedo for layered surfaces can be computed as the weighted sum of the albedos of the individual layers. Non-absorbing clear coat layers can be simply ignored (or the albedo of the perfect specular reflection can be used as well) but absorption should be taken into account.

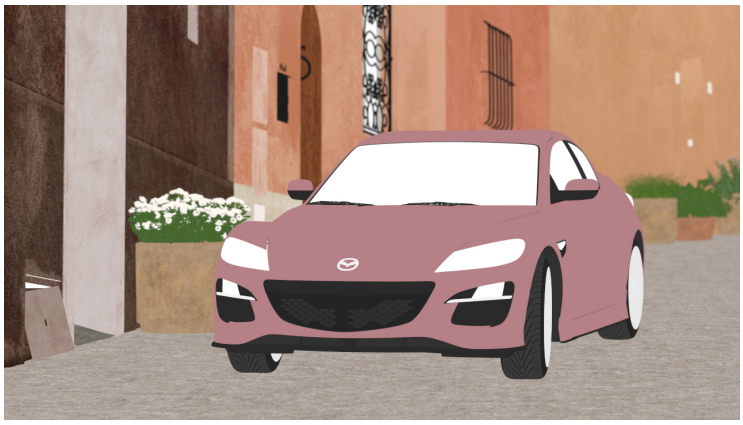

Figure 3.3 – Example albedo image rendered using the first hit. Note that the albedos of all transparent surfaces are 1.

Figure 3.4 – Example albedo image rendered using the first diffuse or glossy (non-delta) hit. Note that the albedos of perfect specular (delta) transparent surfaces are computed as the Fresnel blend of the reflected and transmitted albedos.

Normal

The normal image should contain the shading normals of the surfaces either in world-space or view-space. It is recommended to include normal mapping to preserve as much detail as possible.

Just like any other input image, the normal image should be anti-aliased (i.e. by accumulating the normalized normals per pixel). The final accumulated normals do not have to be normalized but must be in a range symmetric about 0 (i.e. normals mapped to $[0, 1]$ are *not* acceptable and must be remapped to e.g. $[-1, 1]$).

Similar to the albedo, the normal can be stored for either the first or a subsequent hit (if the first hit has a perfect specular/delta BSDF).

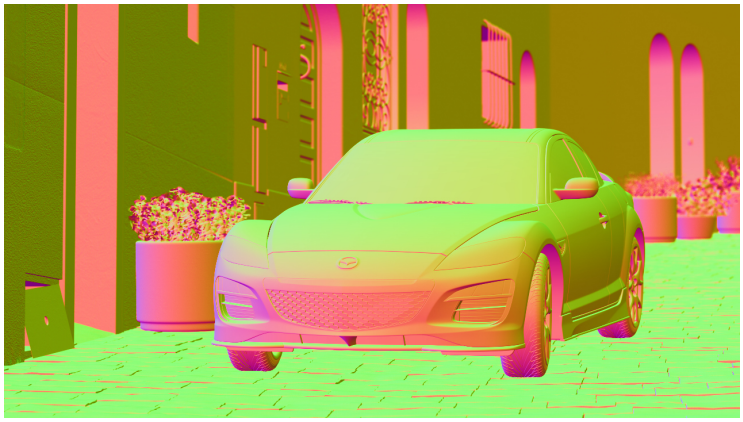

Figure 3.5 – Example normal image rendered using the first hit (the values are actually in $[-1, 1]$ but were mapped to $[0, 1]$ for illustration purposes).

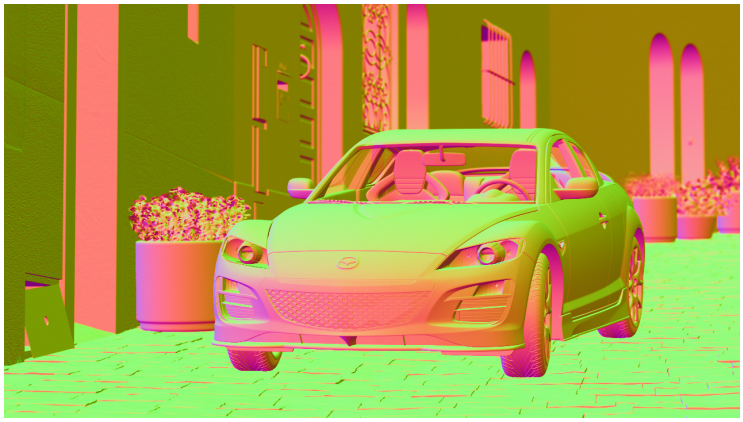

Figure 3.6 – Example normal image rendered using the first diffuse or glossy (non-delta) hit. Note that the normals of perfect specular (delta) transparent surfaces are computed as the Fresnel blend of the reflected and transmitted normals.

Chapter 4
Examples

Denoise

A minimal working example demonstrating how to use Open Image Denoise can be found at `examples/denoise.cpp`, which uses the C++11 convenience wrappers of the C99 API.

This example is a simple command-line application that denoises the provided image, which can optionally have auxiliary feature images as well (e.g. albedo and normal). The images must be stored in the Portable FloatMap (PFM) format, and the color values must be encoded in little-endian format.

Running `./denoise` without any arguments will bring up a list of command line options.