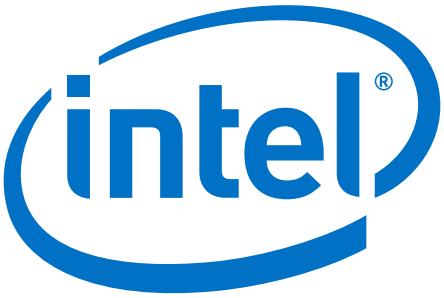

INTEL® OPEN IMAGE DENOISE

HIGH-PERFORMANCE DENOISING LIBRARY FOR RAY TRACING

Version 1.2.3
September 10, 2020

Contents

Chapter 1

Intel Open Image Denoise Overview

Intel Open Image Denoise is an open source library of high-performance, high-quality denoising filters for images rendered with ray tracing. Intel Open Image Denoise is part of the Intel® oneAPI Rendering Toolkit and is released under the permissive Apache 2.0 license.

The purpose of Intel 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 Intel Open Image Denoise library is a collection of efficient deep learning based denoising filters, which were 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.

Although the library ships with a set of pre-trained filter models, it is not mandatory to use these. To optimize a filter for a specific renderer, sample count, content type, scene, etc., it is possible to train the model using the included training toolkit and user-provided image datasets.

Intel 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.

Intel Open Image Denoise internally builds on top of Intel oneAPI Deep Neural Network Library (oneDNN), 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 Intel Open Image Denoise.

Support and Contact

Intel 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 Intel 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`.

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

Version History

Changes in v1.2.3:

- Fixed incorrect detection of AVX-512 on macOS (sometimes causing a crash)
- Fixed inconsistent performance and costly initialization for AVX-512
- Fixed JIT'ed AVX-512 kernels not showing up correctly in VTune

Changes in v1.2.2:

- Fixed unhandled exception when canceling filter execution from the progress monitor callback function

Changes in v1.2.1:

- Fixed tiling artifacts when in-place denoising (using one of the input images as the output) high-resolution (> 1080p) images
- Fixed ghosting/color bleeding artifacts in black regions when using albedo/normal buffers
- Fixed error when building as a static library (`OIDN_STATIC_LIB` option)
- Fixed compile error for ISPC 1.13 and later
- Fixed minor TBB detection issues
- Fixed crash on pre-SSE4 CPUs when using some recent compilers (e.g. GCC 10)
- Link C/C++ runtime library dynamically on Windows too by default
- Renamed example apps (`oidnDenoise`, `oidnTest`)
- Added benchmark app (`oidnBenchmark`)
- Fixed random data augmentation seeding in training
- Fixed training warning with PyTorch 1.5 and later

Changes in v1.2.0:

- Added neural network training code
- Added support for specifying user-trained models at runtime
- Slightly improved denoising quality (e.g. less ringing artifacts, less blurriness in some cases)
- Improved denoising speed by about 7-38% (mostly depending on the compiler)
- Added `OIDN_STATIC_RUNTIME` CMake option (for Windows only)
- Added support for OpenImageIO to the example apps (disabled by default)
- Added check for minimum supported TBB version
- Find debug versions of TBB
- Added testing

Changes in v1.1.0:

- Added `RTLightmap` filter optimized for lightmaps
- Added `hdrScale` filter parameter for manually specifying the mapping of HDR color values to luminance levels

Changes in v1.0.0:

- Improved denoising quality
  - More details preserved
  - Less artifacts (e.g. noisy spots, color bleeding with albedo/normal)
- Added `maxMemoryMB` filter parameter for limiting the maximum memory consumption regardless of the image resolution, potentially at the cost of lower denoising speed. This is internally implemented by denoising the image in tiles
- Significantly reduced memory consumption (but slightly lower performance) for high resolutions (> 2K) by default: limited to about 6 GB
- Added `alignment` and `overlap` filter parameters that can be queried for manual tiled denoising
- Added `verbose` device parameter for setting the verbosity of the console output, and disabled all console output by default
- Fixed crash for zero-sized images

Changes in v0.9.0:

- Reduced memory consumption by about 38%
- Added support for progress monitor callback functions
- Enabled fully concurrent execution when using multiple devices
- Clamp LDR input and output colors to 1
- Fixed issue where some memory allocation errors were not reported

Changes in v0.8.2:

- Fixed wrong HDR output when the input contains infinities/NaNs
- Fixed wrong output when multiple filters were executed concurrently on separate devices with AVX-512 support. Currently the filter executions are serialized as a temporary workaround, and a full fix will be included in a future release.
- Added `OIDN_STATIC_LIB` CMake option for building as a static library (requires CMake 3.13.0 or later)
- Fixed CMake error when adding the library with add_subdirectory() to a project

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

Compiling Intel Open Image Denoise

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

Prerequisites

You can clone the latest Intel Open Image Denoise sources using Git with the Git Large File Storage (LFS) extension installed:

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

Please note that installing the Git LFS extension is *required* to correctly clone the repository. Cloning without Git LFS will seemingly succeed but actually some of the files will be invalid and thus compilation will fail.

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

- 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)
- Intel® SPMD Program Compiler (ISPC), version 1.14.1 or later. Please obtain a release of ISPC from the ISPC downloads page. The build system looks for ISPC in the `PATH` and in the directory right "next to" the checked-out Intel Open Image Denoise sources.[1] Alternatively set the CMake variable `ISPC_EXECUTABLE` to the location of the ISPC compiler.
- Python 2.7 or later
- Intel® Threading Building Blocks (TBB) 2017 or later

[1] For example, if Intel Open Image Denoise is in `~/Projects/oidn`, ISPC will also be searched in `~/Projects/ispc-v1.12.0-linux`

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 on Linux/macOS

Assuming the above prerequisites are all fulfilled, building Intel 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.

Entitlements on macOS

macOS requires notarization of applications as a security mechanism, and entitlements must be declared during the notarization process.
Intel Open Image Denoise uses just-in-time compilaton through the Intel Deep Neural Network Library and requires the following entitlements:

- `com.apple.security.cs.allow-jit`
- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.disable-executable-page-protection`

Compiling on Windows

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

- Browse to the Intel 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 (Intel 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, Intel 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`".

CMake Configuration

The default CMake configuration in the configuration dialog should be appropriate for most usages. The following list describes the options that can be configured in CMake:

- `CMAKE_BUILD_TYPE`: Can be used to switch between Debug mode (Debug), Release mode (Release) (default), and Release mode with enabled assertions and debug symbols (RelWithDebInfo).
- `OIDN_STATIC_LIB`: Build Intel Open Image Denoise as a static library (OFF by default). CMake 3.13.0 or later is required to enable this option. When using the statically compiled Intel Open Image Denoise library, you either have to use the generated CMake configuration files (recommended), or you have to manually define `OIDN_STATIC_LIB` before including the library headers in your application.
- `OIDN_STATIC_RUNTIME`: Use the static version of the C/C++ runtime library (available only on Windows, OFF by default).
- `OIDN_APPS`: Enable building example and test applications (ON by default).

- `OIDN_APPS_OPENIMAGEIO`: Enable OpenImageIO support in the example and test applications to be able to load/save OpenEXR, PNG, and other image file formats (OFF by default).
- `TBB_ROOT`: The path to the TBB installation (autodetected by default).
- `OPENIMAGEIO_ROOT`: The path to the OpenImageIO installation (autodetected by default).

Chapter 3

Intel Open Image Denoise API

Intel 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 Intel 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 Intel 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

Intel 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.

Table 3.2 – Parameters supported by all devices.

Type	Name	Default	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
int	verbose	0	verbosity level of the console output between 0–4; when set to 0, no output is printed, when set to a higher level more output is printed

Table 3.3 – Additional parameters supported only by CPU devices.

Type	Name	Default	Description
int	numThreads	0	maximum number of threads which the library 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

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 Intel 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 Intel 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 Intel 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
OIDN_ERROR_CANCELLED	the operation was cancelled by the user

Buffer

Large data like images can be passed to Intel 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 Intel 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 images, and potentially setting other parameter values as well.

To bind images 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.

Some special data used by filters are opaque/untyped (e.g. trained model weights blobs), which can be specified with the `oidnSetSharedFilterData` function:

```
void oidnSetSharedFilterData(OIDNFilter filter, const char* name,
                             void* ptr, size_t byteSize);
```

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);
void  oidnSetFilter1f(OIDNFilter filter, const char* name, float value);
bool  oidnGetFilter1b(OIDNFilter filter, const char* name);
int   oidnGetFilter1i(OIDNFilter filter, const char* name);
float oidnGetFilter1f(OIDNFilter filter, const char* name);
```

Filters support a progress monitor callback mechanism that can be used to report progress of filter operations and to cancel them as well. Calling `oidnSetFilterProgressMonitorFunction` registers a progress monitor callback function (`func` argument) with payload (`userPtr` argument) for the specified filter (`filter` argument):

```
typedef bool (*OIDNProgressMonitorFunction)(void* userPtr, double n);

void oidnSetFilterProgressMonitorFunction(OIDNFilter filter,
                                          OIDNProgressMonitorFunction func,
                                          void* userPtr);
```

Only a single callback function can be registered per filter, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function. Once registered, Intel Open Image Denoise will invoke the callback function multiple times during filter operations, by passing the payload as set at registration time (`userPtr` argument), and a `double` in the range $[0, 1]$ which estimates the progress of the operation (`n` argument). When returning `true` from the callback function, Intel Open Image Denoise will continue the filter operation normally. When returning `false`, the library will cancel the filter operation with the `OIDN_ERROR_CANCELLED` error code.

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 Intel 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 convolutional neural network (CNN), and it aims to provide a good balance between denoising performance and quality. The filter comes with a set of pre-trained CNN models that work well with a wide range of ray tracing based renderers and noise levels.

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 a 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 parameters listed in the table below. All specified images must have the same dimensions. The output image can be one of the input images (i.e. in-place denoising is supported).

Figure 3.1 – Example noisy color image rendered using unidirectional path tracing (64 spp). *Scene by Evermotion.*

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

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, $+\infty$), 3 channels)
Image	float3	albedo		input feature image containing the albedo (values in [0, 1], 3 channels) 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 ($-\infty$, $+\infty$), 3 channels) of the first hit per pixel; *optional*, requires setting the albedo image too
Image	float3	output		output color image (3 channels); it can be one of the input images
Data		weights		trained model weights blob; *optional*
bool		hdr	false	whether the color is HDR
float		hdrScale	NaN	HDR color values are interpreted such that, multiplied by this scale, a value of 1 corresponds to a luminance level of 100 cd/m^2 (this affects the quality of the output but the output color values will *not* be scaled); if set to NaN, the scale is computed automatically (*default*)
bool		srgb	false	whether the color is encoded with the sRGB (or 2.2 gamma) curve (LDR only) or is linear; the output will be encoded with the same curve
int		maxMemoryMB	6000	approximate maximum amount of scratch memory to use in megabytes (actual memory usage may be higher); limiting memory usage may cause slower denoising due to internally splitting the image into overlapping tiles, but cannot cause the denoising to fail
const int		alignment		when manually denoising the image in tiles, the tile size and offsets should be multiples of this amount of pixels to avoid artifacts; note that manual tiled denoising of HDR images is supported *only* when hdrScale is set by the user
const int		overlap		when manually denoising the image in tiles, the tiles should overlap by this amount of pixels

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.

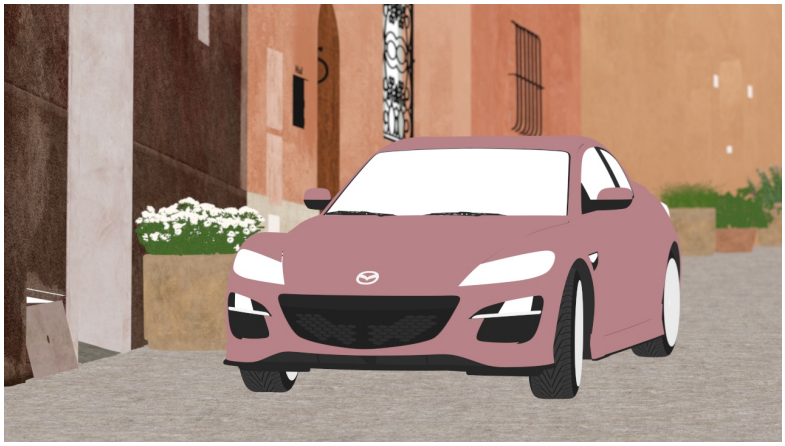

Figure 3.3 – Example albedo image obtained using the first hit. Note that the albedos of all transparent surfaces are 1.

Figure 3.4 – Example albedo image obtained 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.

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 in itself 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.

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 maps 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).

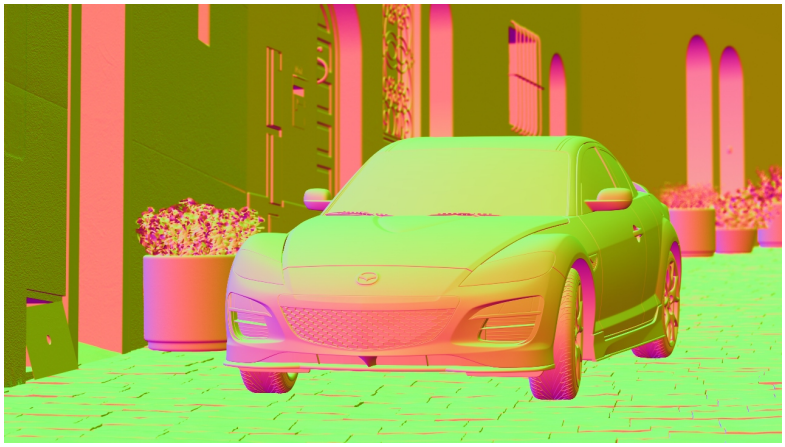

Figure 3.5 – Example normal image obtained using the first hit (the values are actually in $[-1, 1]$ but were mapped to $[0, 1]$ for illustration purposes).

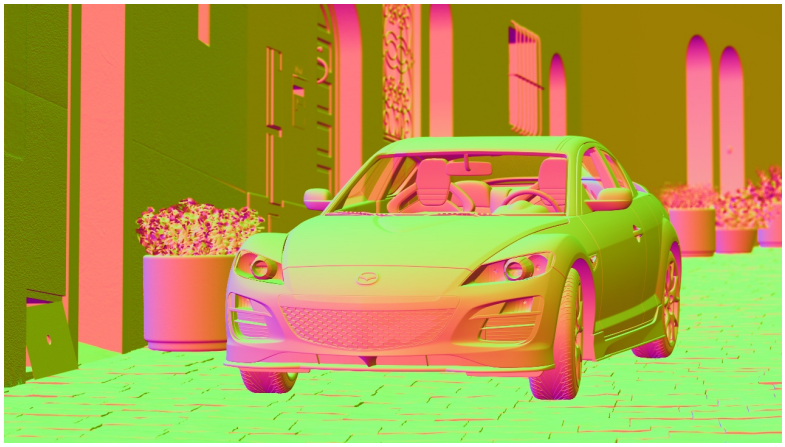

Figure 3.6 – Example normal image obtained using the first diffuse or glossy (non-delta) hit. Note that the normals of perfect specular (delta) transparent surfaces are computed as the Fresnel blend of the reflected and transmitted normals.

Weights

Instead of using the built-in trained models for filtering, it is also possible to specify user-trained models at runtime. This can be achieved by passing the model *weights* blob produced by the training tool.

RTLightmap

The `RTLightmap` filter is a variant of the `RT` filter optimized for denoising HDR lightmaps. It does not support LDR images.

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

Table 3.8 – Parameters supported by the `RTLightmap` filter.

Type	Format	Name	Default	Description
Image	float3	color		input color image (HDR values in $[0, +\infty)$, 3 channels)
Image	float3	output		output color image (3 channels); it can be one of the input images
Data		weights		trained model weights blob; *optional*
float		hdrScale	NaN	HDR color values are interpreted such that, multiplied by this scale, a value of 1 corresponds to a luminance level of 100 cd/m^2; if set to NaN, the scale is computed automatically (*default*)
int		maxMemoryMB	6000	approximate maximum amount of scratch memory to use in megabytes (actual memory usage may be higher)
const int		alignment		when manually denoising the image in tiles, the tile size and offsets should be multiples of this amount of pixels
const int		overlap		when manually denoising the image in tiles, the tiles should overlap by this amount of pixels

Chapter 4
Examples

Denoise

A minimal working example demonstrating how to use Intel Open Image Denoise can be found at `apps/oidnDenoise/oidnDenoise.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). By default the images must be stored in the Portable FloatMap (PFM) format, and the color values must be encoded in little-endian format. To enable other image formats (e.g. OpenEXR, PNG) as well, the project has to be rebuilt with OpenImageIO support enabled.

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

Chapter 5
Training

The Intel Open Image Denoise source distribution includes a Python-based neural network training toolkit (located in the `training` directory), which can be used to train the denoising filter models with image datasets provided by the user. The toolkit consists of the following command-line scripts:

- `preprocess.py`: Preprocesses training and validation datasets.
- `train.py`: Trains a model using preprocessed datasets.
- `infer.py`: Performs inference on a dataset using the specified training result.
- `export.py`: Exports a training result to the runtime model weights format.
- `find_lr.py`: Tool for finding the optimal minimum and maximum learning rates.
- `visualize.py`: Invokes TensorBoard for visualizing statistics of a training result.
- `split_exr.py`: Splits a multi-channel EXR image into multiple feature images.
- `convert_image.py`: Converts a feature image to a different image format.
- `compare_image.py`: Compares two feature images using the specified quality metrics.

Prerequisites

Before you can run the training toolkit you need the following prerequisites:

- Linux (other operating systems are currently not supported)
- Python 3.7 or later
- PyTorch 1.4 or later
- NumPy 1.17 or later
- OpenImageIO 2.1 or later
- TensorBoard 2.1 or later (*optional*)

Datasets

A dataset should consist of a collection of noisy and corresponding noise-free reference images. It is possible to have more than one noisy version of the same image in the dataset, e.g. rendered at different samples per pixel and/or using different seeds.

The training toolkit expects to have all datasets (e.g. training, validation) in the same parent directory (e.g. `data`). Each dataset is stored in its own subdirectory (e.g. `train`, `valid`), which can have an arbitrary name.

The images must be stored in OpenEXR format (`.exr` files), and the filenames must have a specific format but the files can be stored in an arbitrary directory structure inside the dataset directory. The only restriction is that all versions of an image (noisy images and the reference image) must be located in the same subdirectory. Each feature of an image (e.g. color, albedo) must be stored in a separate image file, i.e. multi-channel EXR image files are not supported. If you have multi-channel EXRs, you can split them into separate images per feature using the included `split_exr.py` tool.

An image filename must consist of a name, the number of samples per pixel or whether it is the reference (e.g. `0128spp`, `ref`), the identifier (ID) of the feature (e.g. `hdr`, `alb`), and the file extension (`.exr`). The exact format as a regular expression is the following:

```
.+_([0-9]+(spp)?|ref|reference|gt|target)\.(hdr|ldr|alb|nrm)\.exr
```

The number of samples per pixel should be padded with leading zeros to have a fixed number of digits. If the reference image is not explicitly named as such (i.e. has the number of samples instead), the image with the most samples per pixel will be considered the reference.

The following image features are supported:

Feature	ID	Channels
color (HDR)	`hdr`	3
color (LDR)	`ldr`	3
albedo	`alb`	3
normal	`nrm`	3

Table 5.1 – Supported image features, their IDs, and their number of channels.

The following directory tree demonstrates an example root dataset directory (`data`) containing one dataset (`rt_train`) with HDR color and albedo feature images:

```
data
`-- rt_train
    |-- scene1
    |   |-- view1_0001.alb.exr
    |   |-- view1_0001.hdr.exr
    |   |-- view1_0004.alb.exr
    |   |-- view1_0004.hdr.exr
    |   |-- view1_8192.alb.exr
    |   |-- view1_8192.hdr.exr
    |   |-- view2_0001.alb.exr
    |   |-- view2_0001.hdr.exr
    |   |-- view2_8192.alb.exr
    |   `-- view2_8192.hdr.exr
    |-- scene2_000008spp.alb.exr
    |-- scene2_000008spp.hdr.exr
```

```
|-- scene2_000064spp.alb.exr
|-- scene2_000064spp.hdr.exr
|-- scene2_reference.alb.exr
`-- scene2_reference.hdr.exr
```

Preprocessing (preprocess.py)

Training and validation datasets can be used only after preprocessing them using the `preprocess.py` script. This will convert the specified training (`-t` or `--train_data` option) and validation datasets (`-v` or `--valid_data` option) located in the root dataset directory (`-D` or `--data_dir` option) to a format that can be loaded more efficiently during training. All preprocessed datasets will be stored in a root preprocessed dataset directory (`-P` or `--preproc_dir` option).

The preprocessing script requires the set of image features to include in the preprocessed dataset as command-line arguments. Only these specified features will be available for training. Preprocessing also depends on the filter that will be trained (e.g. determines which HDR/LDR transfer function has to be used), which should be also specified (`-f` or `--filter` option). The alternative is to manually specify the transfer function (`-x` or `--transfer` option) and other filter-specific parameters, which could be useful for training custom filters.

For example, to preprocess the training and validation datasets (`rt_train` and `rt_valid`) with HDR color, albedo, and normal image features, for training the `RT` filter, the following command can be used:

```
./preprocess.py hdr alb nrm --filter RT --train_data rt_train --valid_data rt_valid
```

For more details about using the preprocessing script, including other options, please have a look at the help message:

```
./preprocess.py -h
```

Training (train.py)

After preprocessing the datasets, it is possible to start training a model using the `train.py` script. Similar to the preprocessing script, the input features must be specified (could be a subset of the preprocessed features), and the dataset names, directory paths, and the filter can be also passed.

The tool will produce a training *result*, the name of which can be either specified (`-r` or `--result` option) or automatically generated (by default). Each result is stored in its own subdirectory, and these are located in a common parent directory (`-R` or `--results_dir` option). If a training result already exists, the tool will resume training that result from the latest checkpoint, or from an earlier checkpoint at the specified epoch (`-c` or `--checkpoint` option).

The default training hyperparameters should work reasonably well in general, but some adjustments might be necessary for certain datasets to attain optimal performance, most importantly: the number of epochs (`-e` or `--epochs` option), the mini-batch size (`--bs` or `--batch_size` option), and the learning rate. The training tool uses a cyclical learning rate (CLR) with the `triangular2` scaling policy and an optional linear ramp-down at the end. The learning rate schedule can be configured by setting the base learning rate (`--lr` or `--learning_rate` option), the maximum learning rate (`--max_lr` or `--max_learning_rate` option), and the total cycle size in number of epochs (`--lr_cycle_epochs` option). If there is an incomplete cycle at the end, the learning rate will be linearly ramped down to almost zero.

Example usage:

```
./train.py hdr alb --filter RT --train_data rt_train --valid_data rt_valid --result rt_hdr_alb
```

For finding the optimal learning rate range we recommend using the included `find_lr.py` script, which trains one epoch using an increasing learning rate and logs the resulting losses in a comma-separated values (CSV) file. Plotting the loss curve can show when the model starts to learn (the base learning rate) and when it starts to diverge (the maximum learning rate).

The model is evaluated with the validation dataset at regular intervals (`--valid_epochs` option), and checkpoints are also regularly created (`--save_epochs` option) to save training progress. Also, some statistics are logged (e.g. training and validation losses, learning rate) at a specified frequency (`--log_steps` option), which can be later visualized with TensorBoard by running the `visualize.py` script, e.g.:

```
./visualize.py --result rt_hdr_alb
```

Inference (infer.py)

A training result can be tested by performing inference on an image dataset (`-i` or `--input_data` option) using the `infer.py` script. The dataset does *not* have to be preprocessed. In addition to the result to use, it is possible to specify which checkpoint to load as well. By default the latest checkpoint is loaded.

The tool saves the output images in a separate directory (`-O` or `--output_dir`) in the requested formats (`-F` or `--format` option). It also evaluates a set of image quality metrics (`-M` or `--metric` option), e.g. PSNR, SSIM, for images that have reference images available. All metrics are computed in tonemapped non-linear sRGB space. Thus, HDR images are first tonemapped (with Naughty Dog's Filmic Tonemapper from John Hable's *Uncharted 2: HDR Lighting* presentation) and converted to sRGB before evaluating the metrics.

Example usage:

```
./infer.py --result rt_hdr_alb --input_data rt_test --format exr png --metric ssim
```

Exporting Results (export.py)

The training result produced by the `train.py` script cannot be immediately used by the main library. It has to be first exported to the runtime model weights format, a *Tensor Archive* (TZA) file. Running the `export.py` script for a training result (and optionally a checkpoint) will create a binary `.tza` file in the directory of the result, which can be either used at runtime through the API or it can be included in the library build by replacing one of the built-in weights files.

Example usage:

```
./export.py --result rt_hdr_alb
```

Image Conversion and Comparison

In addition to the already mentioned `split_exr.py` script, the toolkit contains a few other image utilities as well.

`convert_image.py` converts a feature image to a different image format (and/or a different feature, e.g. HDR color to LDR), performing tonemapping and other transforms as well if needed. For HDR images the exposure can be adjusted by passing a linear exposure scale (`-E` or `--exposure` option). Example usage:

```
./convert_image.py view1_0004.hdr.exr view1_0004.png --exposure 2.5
```

The `compare_image.py` script compares two feature images (preferably having the dataset filename format to correctly detect the feature) using the specified image quality metrics, similar to the `infer.py` tool. Example usage:

```
./compare_image.py view1_0004.hdr.exr view1_8192.hdr.exr --exposure 2.5 --metric mse ssim
```