NVIDIA

NVIDIA Iray

Programmer's Manual

3 July 2023
Version 2.0

NVIDIA Iray – Programmer's Manual

Copyright Information

Document build number 367100.3485

Contents

Preface

Purpose of this document

This document is the companion for the *NVIDIA Iray® API reference* [IrayAPI] (page 197). This document describes fundamental technical principles underlying NVIDIA Iray rendering technology. References to neuray® in the documentation refer to the library of functions used by Iray.

Audience

This document is intended for application and system developers who intend to integrate NVIDIA Iray rendering technology in their applications. This document presumes advanced programming skills in C++ and familiarity with 3D rendering concepts. No specialized programming skills in rendering technology and networking are required.

How this document is organized

This document is organized as follows:

- "Getting started" (page 3) provides an overview of release contents, the example programs, the steps to compile and run a program, and render an image. The example programs are a good place to start if you prefer to learn by modifying working examples.
- "Library design" (page 11) provides an overview of neuray which is the library of functions used by the Iray API, the configuration of the neuray library, and an overview of the Iray scene database where all scene data is stored. It is recommended that you refer back to this section as you explore the neuray library and gain a better understanding of its capabilities.
- Rendering (page 29) describes rendering concepts specific to Iray. "Iray Photoreal" (page 43) and "Iray Interactive" (page 62) describe the Iray rendering modes. Explore these sections in detail. They are key to understanding the capabilities of the Iray rendering modes.
- "Scene database" (page 76) provides brief descriptions and links to some of the more important scene elements and describes how to create a scene programmatically. "Geometry" (page 80), "Materials" (page 90), "Decals" (page 97), "Sources of light" (page 113), "Environment dome" (page 119), and "Camera" (page 124) describe specific scene elements in detail. Because Iray is a physically-plausible renderer, it is important to understand how scenes should be set up to get the best results. Creating optimal scene structures is covered in "Physically plausible scene setup" (page 134).
- "Networking overview" (page 150) describes the networking support provided by Iray. "Networking modes" (page 150) describes the supported networking modes, the advantages and disadvantages of each, and their configuration. "Iray Bridge" (page 151) describes a client-server architecture that enables simple and efficient use of resources of

a remote cluster over a TCP/IP connection. "Server-based rendering" (page 165) describes how to use HTTP and RTMP servers to deliver rendered images to a client.

- "Customization" (page 169) describes how to customize Iray with user-defined classes and plugins.

API symbols and namespaces

API symbols that are qualified by the `mi::neuraylib` namespace are abbreviated; only the unqualified symbol is displayed. In the HTML version of this document,[1] the API symbols are links to their definition in the API documentation. The Links in color button in the navigation bar at the top of the webpage will make linked symbols and phrases more visible.

1. `https://raytracing-docs.nvidia.com/`

1 Getting started

The intention of the following sections is to enable you to quickly get started using Iray. After you finish this chapter, you should know how to:

- Test your Iray installation
- Start-up and shutdown the neuray library used by the Iray API
- Compile and run an example program that renders an image

1.1 Iray release components

The Iray release contains the following parts:

- Installation instructions.
- A platform-dependent shared library, `libneuray`, that is used by the Iray API. The supported platforms are Linux, macOS, and Windows.
- A set of platform-independent C++ include files.
- A library of documents that describe:
  - The Iray API and how to use it.
  - The Material Definition Language (MDL), the language that creates appearance descriptions used by the renderer.
- Collections of:
  - Example programs that illustrate key features of Iray and their implementation.
  - Example materials created with MDL.
  - Example textures.
- End user license agreement.

1.2 Example programs

The following example programs are shipped with the Iray release. They illustrate, in a tutorial manner, the important interfaces and concepts. For compilation instructions, see "How to compile a program" (page 4).

If your variant of the neuray library requires an authentication key you find the details of how to use the key with the example programs in "Starting and shutting down the Iray API" (page 5).

The example programs make use of a shared header file, `example_shared.h` (page 306), which is explained in more detail in "Starting and shutting down the Iray API" (page 5).

- Example – Start-up and shutdown (page 7)

- Example – Importing and rendering a scene file (page 10)
- Example – Configuring the Iray API (page 25)
- Example – Progressively rendering a scene (page 33)
- Example – Starting the API in networking mode (page 151)
- Example – Creating a scene through the API (page 79)
- Example – Using MDL materials and functions (page 96)
- Example – Using triangle meshes (page 81)
- Example – Adding a polygon mesh to a scene (page 82)
- Example – Using on-demand meshes (page 82)
- Example – Using subdivision surfaces (page 83)
- Example – Using free-form surfaces (page 84)
- Example – Creating fibers (page 87)
- Example – Creating particles (page 89)
- Example – Running an HTTP server (page 165)
- Example – Running an interactive video-streaming server (page 168)
- Example – A user-defined class (page 170)
- Example – Creating a plugin (page 171)
- Example – Creating an importer for a custom file format (page 173)
- Example – Creating an exporter for a custom file format (page 174)
- Example – Creating an image exporter for Photoshop (page 174)
- Example – Rendering a scene with Iray Bridge (page 163)

1.3 How to compile a program

This section describes how to compile the example programs shipped with Iray. For a complete list of example programs see "Example programs" (page 3).

1.3.1 Contents of the release

The neuray library release contains a set of example programs. It is recommended that you compile them and take them as a starting point for your own development.

The release contains a `Makefile` to build the examples on Linux and MacOS platforms as well as solution files for Visual Studio 2012 to build the examples on a Microsoft Windows platform.

You can integrate the neuray library easily in other build environments. You can even compile the examples by hand following the steps below. Let the environment variable `$IRAY_ROOT` refer to the installation root of the neuray library.

1.3.2 Procedure

To compile an example program:

1. Copy the examples shipped with the release, make them writable, and switch to that directory:

```
cp -r $IRAY_ROOT/examples .
chmod -R u+w examples
cd examples
```

2. Compile the program with the appropriate include path switch that points the compiler to the location of the neuray include files. A g++ compiler call could look like this:

```
g++ -I$IRAY_ROOT/include -c example_start_shutdown.cpp -o ↦
    example_start_shutdown.o -pthread
```

3. Link the program with the library that provides `dlopen()`. A `g++` linker call could look like this:

```
g++ -o example_start_shutdown example_start_shutdown.o -ldl -↦
    pthread
```

4. Make the shared library, also known as a DLL, known to your runtime system so that it is found when starting the example program. You can do this by placing the shared library in an appropriate location, and/or setting environment variables such as `PATH` (on Windows), `LD_LIBRARY_PATH` (on Linux), or `DYLD_LIBRARY_PATH` and `DYLD_FRAMEWORK_PATH` (on macOS).

> **Note:** This step is platform and installation dependent.

1.4 How to run a program

This section describes how to run the example programs shipped with Iray. For a complete list of example programs see "Example programs" (page 3), and how to compile them (page 4).

The example programs that come with the neuray library release are programmed to run on a command line. Some examples expect additional parameters and they will show a help message in case parameters are missing.

Some programs use additional files or produce files in the directory as a result of their processing.

Please refer to the comments in the example source code and the respective section that describes an example for details.

The example programs report results as well as error messages on stdout and stderr to the terminal where they are running. On Windows, the helper function keep_console_open() prevents the terminal from closing when running with an attached debugger, for example, when running within Visual Studio. This helper function is defined in the `example_shared.h` (page 306) header file that is used throughout the examples.

1.5 Starting and shutting down the Iray API

This section introduces:

1. The neuray object (page 6), the access point for the system.
2. The required interfaces and methods:
   - The `INeuray` (page 7) interface to start and shutdown the Iray API.
   - Function `mi::mi_factory` (page 6) to create the neuray object.
   - Function `INeuray::get_status` (page 7) to query the status of the Iray API.
3. The program `example_start_shutdown.cpp` (page 7), which demonstrates how to start up and shut down the Iray API
4. File `example_shared.h` (page 7), a shared example header file, which provides common functionality for all example programs

To access the Iray API, you create a neuray object using a factory function. This object is your only access point to the system. You use it to authenticate your application against the neuray library, configure the system, start it up and shut it down. Only one instance of this object can exist at any time.

When startup is completed, you can build scenes and initiate renderings. Before you can shut down the Iray API, all render jobs must be finished and all transactions must be closed.

See "Runtime configuration" (page 24) for available configurations and "Logging configuration" (page 24) for an example configuring the log output.

1.5.1 Loading and accessing the neuray library

The factory function, which is used to create the neuray object, is a C function which returns an interface pointer to the neuray object. This C function is the only exported global function of the neuray library:

```
mi_factory()
```

This function has actually two parameters, one to configure a memory allocator and one to guarantee the correct API version. Both parameters have default values and the examples use this function without special arguments.

This allows you to easily use neuray as a shared library, which can be loaded on demand without requiring to statically link an interface library to the application. The details of loading a shared library on demand and calling this factory function are operating system specific. Since this functionality is needed in all examples, it has been factored into a separate convenience function, load_and_get_ineuray(), and placed in the shared header file `example_shared.h` (page 7), described below.

In addition, this convenience function handles authentication and error situations with some rudimentary reporting. This function can serve as a starting point for applications, but should be reviewed and customized to the particular needs of an application where needed.

To unload the neuray library at the end of a program, you can use the `unload()` function provided in the same header file.

1.5.2 Authenticating an application against the neuray library

The use of the neuray library requires a valid license. Depending on the license some functionality might not be available. A missing or invalid license might cause rendered

images to show a watermark. The example programs will work without a license to the extent that they might show the watermark in images or indicate with a message that functionality is missing.

For most versions of the neuray library, an application needs to authenticate itself against the neuray library that it is in the possession of a license. This authentication is also handled in the load_and_get_ineuray() convenience function. For it to work, you need to place your license key next to the example programs. The license key can come in one of two forms:

1. An authentication.h file, which you can drop as a replacement of the same file in the examples source directory and recompile. This is also the recommended way for applications to compile the license into the application.
2. An examples.lic file, which you can drop into the compiled example programs directory and it will be picked up at runtime. This is only recommended for demo or otherwise limited licenses.

After you have obtained successfully the neuray object and authenticated the application, you can move on to configure the API if needed, which is skipped here, and finally start the Iray API.

1.5.3 Start-up and shutdown

The `INeuray` interface is used to start and shut down the Iray API. Most of the API can only be used after it has been started (and before it has been shut down). Startup does not happen during the `mi_factory` call because you might want to configure the behavior of the API, which for certain configurations has to happen before startup. For details, see "Runtime configuration" (page 24) and "Logging configuration" (page 24).

The status of the API can be queried using the `INeuray::get_status` method.

Finally, you have to shut down the Iray API. At this point, you should have released all interface pointers (page 17) except the pointer to the main `INeuray` interface. If you are using the Handle class (page 20), make sure that all handles have gone out of scope.

1.5.4 Example – Start-up and shutdown

Source code
`example_start_shutdown.cpp` (page 309)

This example program accesses the main neuray interface, queries the API interface version, prints diagnostic information about the API interface version, API version and build version string, and then starts and shuts down the neuray API. The example illustrates also the necessary places for error checking.

See also "How to compile a program" (page 4) and "How to run a program" (page 5).

1.5.5 Example – Shared header file

Source code
`example_shared.h` (page 306)

The shared example header file provides the following common functionality for all example programs:

`load_and_get_ineuray()`
Described in "Loading and accessing the neuray library" (page 6)

`unload()`
Unloads the neuray library.

`keep_console_open()`
Prevents the terminal to close when the program is executed with an attached debugger under Windows, most notably, from within Visual Studio. It is void on other platforms.

`check_success(expr)`
A macro, which, similar to an assertion, checks the conditional expression `expr` and exits the program with a diagnostic message if this expression does not evaluate to true.

`sleep_seconds(int seconds)`
Waits for the indicated time.

`snprintf()`
Can be used across platforms without warnings.

1.6 A first image

This section introduces:

1. Core concepts about rendering images with Iray:
   - Configuration (page 8)
   - Accessing the database (page 9)
   - Importing a scene file (page 9)
   - Rendering a scene (page 9)
2. Example source code file `example_rendering.cpp` (page 10), which demonstrates how to render an image with Iray.

To render a first image, you need to follow the steps explained in "Starting and shutting down the Iray API" (page 5) to obtain the neuray object. Before actually starting the Iray API, you configure it – in this example you provide the material search path and load plugins – and after starting the Iray API you load the scene from a file into the database, render the image, and write the result into an image file.

1.6.1 Configuration

The minimal configuration needed for this example is the search path setting and the plugin loading.

When loading assets like textures or materials, Iray uses search paths to locate those assets, This example imports .mi files (page 178), which reference MDL materials, written in the NVIDIA Material Definition Language (MDL) (page 90). The search path to find those materials is set here. For this example, the search path is actually provided as one of the input parameters to the example program.

Iray makes itself use of plugins to provide different functionality. This example needs to load the OpenImageIO plugin to support various image formats, the .mi importer plugin to load

the scene in this file format, and the `libiray` plugin to render with the Iray Photoreal render mode.

1.6.2 Accessing the database

The `IDatabase` interface is one of the central interface classes that become available once neuray has been started. Using this interface, you can access the scene database (page 76). In particular, you can either access the global scope (page 23) or create your own scope (page 23). The `IScope` interface allows you to create transactions (page 23), which are required for all operations that access the database.

This example uses the database interface to access the global scope and creates a transaction in the global scope. All transactions need either to get committed or aborted. By committing a transaction, all changes (if any) will become visible for subsequently started transactions. Aborting a transaction discards any changes made in that transaction.

1.6.3 Importing a scene file

The import of files is handled by the API component `IImport_api`. The actual import is done by the method `IImport_api::import_elements`(). The result of the import operation is described by the interface `IImport_result`. This interface provides among other things the method `IImport_result::get_error_number`() to retrieve the resulting error number. This error number is 0 in case of success, all other numbers indicate failures.

The example also creates an instance of the `IScene` interface, which is later used for rendering. To this end, the name of the root group, the camera, and the options are set in the scene. These names are available from the import result.

1.6.4 Rendering a scene

Rendering a scene requires a *render mode*, which might require the loading of a corresponding plugin, a *render target* and a *render context*.

A list of render modes with their plugin names and string value name, for the use with the render context below, is shown here (page 32). For the example program below, the chosen render mode is Iray Photoreal with the required `libiray` plugin and the string value name `iray` (all lowercase).

To render a scene, you have to provide a render target with space for the rendered images. In principle, a single render call can render several images at once, each stored in a *canvas.* Canvases have names and these names determine the actual images render, for example, the final beauty pass result, alpha channel, depth or normal pass. The relevant API interfaces are `IRender_target`, `ICanvas`, and `ITile`. However, the Iray API provides a default canvas class implementation, which is used in the example below. Furthermore, the implementation of a render target has been factored into a separate header file, example_render_target_simple.h, which is used across several examples. More details about customizing a render target of canvas class can be found in the next section (page 10).

The render context can be obtained from `IScene::create_render_context`(). Here, the actual render mode (page 32) is selected with its string value name, which is `iray` in the example program below.

1.6.5 Example – Importing and rendering a scene file

Source code

`example_rendering.cpp` (page 282)

This example program expands the `example_start_shutdown.cpp` (page 309) example program and adds a `configuration()` and a `rendering()` function. The `configuration()` function sets the MDL search path and loads the required plugin libraries. The `rendering()` function goes through all the other steps described above; it imports a scene file, renders the scene and writes the resulting image to disk.

This example program expects two command line arguments; the search path for the MDL materials and the file name for the scene file.

2 Library design

The following sections provide an introduction to the library design.

2.1 API modules

The Iray API is based on and uses the Base API and the Math API.

Base API
: provides basic types, functions on them, assertions, and configurations.

Math API
: provides vector, matrix, bbox, and color classes and math functions.

Iray API
: is a rendering component that you can embed in your application to provide advanced rendering functionality for your application. It includes an advanced distributed database with incremental scene updates and scalable rendering on large distributed clusters.

All three APIs use *modules* to structure their presentation in the reference documentation.

2.2 Shared library

The Iray API is a C++ API. Its implementation is provided to you in the neuray library, which is a dynamic library (a.k.a. shared library or DLL), named `libneuray`, with the usual platform specific file extension.

The main benefits of the dynamic library are:

- You can link the neuray library to your application or dynamically load it on demand at runtime without requiring to statically link an interface library to the application. This can improve application startup time.
- The Iray API follows conventional modern C++ library design principles for component software to achieve binary compatibility and future extensibility across shared library boundaries. The neuray library relies therefore only on the stable C ABI of a platform and you can use a different compiler for the application then the compiler used for the neuray library.
- Iray API and ABI versioning allows the safe upgrade of the neuray shared library. For details about when a recompilation is necessary and when not. see "Version numbers and compatibility" (page 16).

Note: The neuray system itself depends on other shared libraries. Several of those are also only loaded on demand as plugins when needed. See `IPlugin_configuration` for details and "A first image" (page 8) for an example.

2.3 Naming conventions

This section defines the terms Iray API and neuray and the naming conventions used by neuray.

Product name
Iray API refers to the API shipped with Iray

Library name
neuray refers to the library of functions used by the Iray API.

The neuray library is written in C++. It uses the following naming conventions:

Symbol type	*Naming convention*
Namespaces	Use `mi` and `mi::neuraylib` for neuray identifiers
Macro names	Use `MI_NEURAYLIB_` as a prefix
Identifier names	Use _ to concatenate multiple words
Function names	Written in lowercase only
Type and class names	Start with one initial uppercase letter
Interface class names	Start with an additional capital “I”.

2.4 Include files

The Iray API offers two groups of include files. One group consists of the `Main include files` while the other group consists of the `Sub-include files`.

Include files are self-contained. Internal dependencies are resolved automatically. As a consequence, their inclusion is order independent.

2.4.1 Main include files

The `Main include files` provide the full functionality of their respective API. They do so by including all necessary `Sub-include files`.

Main include file	*Description*
`mi/base.h`	Includes all `Sub-include files` of the Base API.
`mi/math.h`	Includes `mi/base.h` and all sub-include files of the Math API.
`mi/neuraylib.h`	Includes `mi/base.h`, `mi/math.h` and all sub-include files of the Iray API. Simply including this header gives you access to all the functionality of the Iray API.

2.4.2 Sub-include files

The sub-include files provide individual parts of their respective API grouped by functionality. If you include the respective main include file of an API, then there is no need to include sub-include files of that API anymore and of those APIs it relies on.

There may be include files that are not documented here. They are automatically included if needed and their independent use is not supported. Their naming and organization might change in the future.

See:

- `Base API Include Files` for the sub-include files of the Base API.
- `Math API Include Files` for the sub-include files of the Math API.
- `Iray SDK API Include Files` for the sub-include files of the Iray API.

2.5 Include file compile-time configuration

The Iray API offers a couple of options to configure the compile time behavior of the include files. Since the Iray API is based on the Base API and Math API, their configuration options are also relevant here. See:

- Base API configuration options for assertions (page 13)
- Base API configuration options for STL use (page 13)
- Math API configuration options for assertions (page 14)
- Iray API configuration options for assertions (page 15)

2.5.1 Base API configuration options for assertions

The Base API supports quality software development with assertions. They are contained in various places in the Base API include files.

These tests are switched off by default to have the performance of a release build. To activate the tests, you need to define the two macros `mi::base::mi_base_assert` and `mi::base::mi_base_assert_msg` before including the relevant include files. Defining only one of the two macros is considered an error. These macros and their parameters have the following meaning:

`mi::base::mi_base_assert(expr)`
: If `expr` evaluates to `true` this macro shall have no effect. If `expr` evaluates to `false` this macro may print a diagnostic message and change the control flow of the program, such as aborting the program or throwing an exception. But it may also have no effect at all, for example if assertions are configured to be disabled.

`mi::base::mi_base_assert_msg(msg)`
: Same behavior as `mi::base::mi_base_assert`, but the `msg` text string contains additional diagnostic information that may be shown with a diagnostic message. Typical usages would contain `precondition` or `postcondition` as clarifying context information in the `msg` parameter.

See the documentation for `mi::base::mi_base_assert` in the Base API documentation.

2.5.2 Base API configuration options for STL use

The Base API makes occasional use of Standard Template Library (STL) headers, classes, and functions. This behavior can be customized with the following macro:

`MI_BASE_NO_STL`
The Base API will not include any STL header and it will not use any STL classes not functions. Instead it will add missing class and function definitions to its own namespace, for example `min` and `max` function templates.

> **Note:** The Base API still includes a few standard C++ headers, for example `cstddef` and `cmath`.

The Base API locates classes and functions from the standard library in the namespace scope MISTD. It is by default set as a namespace alias to point to the namespace std. This behavior can be customized with the following macros:

`MI_BASE_STD_NAMESPACE`
This macro has to expand to a valid namespace scope that contains the standard library classes and functions. It is used for the namespace alias `MISTD` instead of the default std.

`MI_BASE_NO_MISTD_ALIAS`
If this macro is defined, the namespace alias definition for `MISTD` is suppressed. This can be used if the standard library classes and functions have been configured to live already in the `MISTD` namespace.

2.5.3 Math API configuration options for assertions

The Math API configuration uses the Base API configuration.

See:

- Base API configuration options for assertions (page 13)
- Base API configuration options for STL use (page 13)

In particular, the Math API assertions are by default mapped to the Base API assertions.

The Math API supports quality software development with assertions. They are contained in various places in the Math API include files.

These tests are mapped to corresponding Base API assertions by default, which in turn are switched off by default to have the performance of a release build. To activate the tests, you need to define the two macros `mi_math_assert` and `mi_math_assert_msg` before including the relevant include files. Defining only one of the two macros is considered an error. These macros and their parameters have the following meaning:

`mi::math::mi_math_assert(expr)`
If `expr` evaluates to `true` this macro shall have no effect. If `expr` evaluates to `false` this macro may print a diagnostic message and change the control flow of the program, such as aborting the program or throwing an exception. But it may also have no effect at all, for example if assertions are configured to be disabled.

`mi::math::mi_math_assert_msg(msg)`
Same behavior as `mi::math::mi_math_assert`, but the `msg` text string contains additional diagnostic information that may be shown with a diagnostic message. Typical usages would contain `precondition` or `postcondition` as clarifying context information in the `msg` parameter.

See `mi::math::mi_math_assert` and `mi::base::mi_base_assert`.

2.5.4 Iray API configuration options for assertions

The Iray API configuration uses the Base API configuration. In addition, the Math API configuration is relevant for the Math API elements used in Iray. For more details, see:

- Base API configuration options for assertions (page 13)
- Base API configuration options for STL use (page 13)
- Math API configuration options for assertions (page 14)

By default, the Iray API assertions are mapped to the Base API assertions.

The Iray API supports quality software development with assertions. They are contained in various places in the Iray API include files.

These tests are mapped to corresponding Base API assertions by default, which in turn are switched off by default to have the performance of a release build. To activate the tests, you need to define the two macros `mi_neuray_assert` and `mi_neuray_assert_msg` before including the relevant include files. Defining only one of the two macros is considered an error. These macros and their parameters have the following meaning:

`mi_neuray_assert(expr)`
: If `expr` evaluates to `true` this macro shall have no effect. If `expr` evaluates to `false` this macro may print a diagnostic message and change the control flow of the program, such as aborting the program or throwing an exception. But it may also have no effect at all, for example if assertions are configured to be disabled.

`mi_neuray_assert_msg(msg)`
: Same behavior as mi_neuray_assert, but the `msg` text string contains additional diagnostic information that may be shown with a diagnostic message. Typical usages would contain `precondition` or `postcondition` as clarifying context information in the `msg` parameter.

See `mi_neuray_assert` and `mi::base::mi_base_assert`.

2.6 Main API access point

To access the Iray API, you create a neuray object using the `mi::mi_factory` function:

Listing 2.1

```
mi::base::IInterface* mi::mi_factory(const mi::base::Uuid& iid);
```

This factory function is the only public access point to all algorithms and data structures in the Iray library. It returns a pointer to an instance of the class identified by the given `UUID`.

The `mi::mi_factory` function supports the following interfaces:

- An instance of the main `INeuray` interface, which is used to configure, to start up, to operate and to shut down the Iray API. This interface can be requested only once.
- An instance of the `IVersion` class.

`mi_factory` allows you to easily use neuray as a shared library, which can be loaded on demand without requiring to statically link an interface library to the application. The details of loading a shared library on demand and calling this factory function are specific to the

operating system. Since this functionality is needed in all examples, it has been factored into a separate convenience function, `load_and_get_ineuray`, and placed in the shared header file `example_shared.h` (page 7) described in "Starting and shutting down the Iray API" (page 5).

For the underlying rationale of a single access function returning an interface pointer, see also "Interfaces" (page 17).

> **Note:** It is not required to use `load_and_get_ineuray`. In particular in larger setups you might want to write your own code to load the neuray DSO and to locate the factory function. In such cases, you call `mi_factory` directly. For simplicity, the examples will use the convenience function `load_and_get_ineuray` instead of `mi_factory`

2.7 Version numbers and compatibility

There are several aspects of the API that may change over time. Version numbers are used to document various kinds of compatibility between different versions.

2.7.1 API Version of a shared library

Assuming the API is provided in the form of a shared library (also known as DLL), you may be able to upgrade the API to a new version without recompiling your application and also without relinking your application, just by replacing the old shared library with a new one. There might be API releases that contain only static libraries, or no libraries at all. In both cases the discussion in this section does not apply to you.

The interoperability between your application and the shared library is versioned with the API version number and the following protocol. In your application you ask the shared library for a particular API version by passing the API version number as an argument to the interface querying function `mi::mi_factory`. The shared library then returns either an interface that conforms to this version or, if this version is not supported, a value of `NULL`.

The API version refers to a particular collection of interfaces and their APIs. It is a single integer number that increases over time. The current API version number is provided with the symbolic name MI_NEURAYLIB_API_VERSION, which is the default argument to the interface querying function.

The shared library always supports its own API version number and, if possible, older API version numbers in order to support upgrades from an older shared library version to a newer version. Older API versions of libraries with the same major product version number (see next section) are always supported, while older API versions of libraries with a different major API version are not guaranteed to be supported. Such support will be documented in the Release Notes.

2.7.2 Major and minor API product version

The product version number for the API governs the compatibility on the source code level, that is, the include files and to some extent on the binary level of the shared library. See the previous section for details on binary compatibility.

Iray API has a product version number, which consists of three components: major version number, minor version number, and version qualifier.

Major version number
Changes in the major version number indicate a disruptive change without guaranteeing compatibility with prior versions. Details are documented in the Release Notes. See MI_NEURAYLIB_VERSION_MAJOR.

Minor version number
Changes in the minor version number, where the major version number remains unchanged, indicate an upwards compatible change. Old sources compile without changes. See MI_NEURAYLIB_VERSION_MINOR.

Version qualifier
A string that indicates special versions, such as alpha and beta releases. Final end-customer releases have an empty version qualifier. The compatibility guarantees described here are only given for end-customer releases and do not hold for new features introduced in alpha or beta releases. See MI_NEURAYLIB_VERSION_QUALIFIER.

See also:

- `Base API Version Numbers and Compatibility`
- `Math API Version Numbers and Compatibility`

2.8 Interfaces

The Iray API follows conventional modern C++ library design principles for component software to achieve binary compatibility across shared library boundaries and future extensibility. The design provides access to the shared library through interfaces, abstract base classes with pure virtual member functions. Parameters of those interface methods are interface pointers or simple types and POD types, such as the math vector class.

The global function `mi::mi_factory` returns the main interface `INeuray` that allows access to the whole library. From this interface other interfaces of the library can be accessed with the `INeuray::get_api_component` member function.

See also "Reference counting" (page 17) and "Handle class" (page 20).

2.9 Reference counting

In neuraylib, interfaces (page 17) implement reference counting for life-time control. Whenever a function returns a pointer to `mi::base::IInterface` or a subclass thereof, the corresponding reference counter has already been increased by 1. That is, you can use the interface pointer without worrying whether the pointer is still valid. Whenever you do not need an interface any longer, you have to release it by calling its release method. Omitting such calls leads to memory leaks.

In more detail, the rules for reference counting are as follows:

- A method that returns an interface increments the reference count for the caller. The caller needs to release the interface when it is done. See Example 1 (page 18), below.
- When a caller passes an interface as method argument the caller has to guarantee that the interface is valid for the runtime of that method, and the callee can safely use the interface for that time without the need to change the reference count.

- If the callee wants to reference an interface passed as an argument later (typically via a member variable) then it has to use reference counting to ensure that the interface remains valid. See Example 2 (page 18), below.

  The initial reference count after construction is 1. Methods returning an interface pointer can therefore be implemented with:

  ```
  return new Foo();
  ```

- Interfaces passed as out arguments of methods are treated similar to return values. The callee decrements the reference count for the value passed in and increments it for the value passed back.

See also "Handle class" (page 20).

2.9.1 Example 1

Assume there is an interface `IFoo` derived from `mi::base::IInterface`, and an API method that creates instances of `IFoo`, for example,

Listing 2.2

```
class IFactory : public mi::base::Interface_declare...>
{
public:
    virtual IFoo* create_foo() = 0;
};
```

Let `factory` be an instance of `IFactory`. As described by the rules above the implementation of create_foo() increments the reference count for the caller. When you are done with your reference to the instance of `IFoo`, you need to decrement its reference count again via `mi::base::IInterface::release`().

Listing 2.3

```
IFoo* foo = factory->create_foo();
// Use "foo"
foo->release();
// Must no longer use "foo" here
```

2.9.2 Example 2

Assume you want to implement the following interface.

Listing 2.4

```
class IRegistry : public mi::base::Interface_declare...>
{
    public:
    virtual void register_foo(IFoo* foo) = 0;
};
```

Further assume that the implementation of `IRegistry` needs to reference the registered instance of `IFoo` after the method `register_foo()` returned. This can be done as follows. Checking `foo` and `m_foo` for `NULL` has been omitted for brevity.

Listing 2.5

```
class Registry_impl : public mi::base::Interface_implement<IRegistry>
{
public:
    Registry_impl(IFoo* foo);
    Registry_impl(const Registry_impl& other);
    Registry_impl& operator=(const Registry_impl& rhs);
    ~Registry_impl();
    void register_foo(IFoo* foo);
private:
    IFoo* m_foo;
};

Registry_impl::Registry_impl(IFoo* foo)
{
    m_foo = foo;
    m_foo->retain();
}

Registry_impl::Registry_impl(const Registry_impl& other)
  : mi::base::Interface_implement<IRegistry>(other)
{
    m_foo = other.m_foo;
    m_foo->retain();
}

Registry_impl& Registry_impl::operator=(const Registry_impl& rhs)
{
    mi::base::Interface_implement<IRegistry>::operator=(rhs);
    m_foo->release();
    m_foo = rhs.m_foo;
    m_foo->retain();
    return *this;
}

Registry_impl::~Registry_impl()
{
    m_foo->release();
}

void Registry_impl::register_foo(IFoo* foo)
{
    m_foo->release();
    m_foo = foo;
```

```
    m_foo->retain();
}
```

As described by the rules above you need to increment and decrement the reference count of the `IFoo` instance if you keep a reference that exceeds the lifetime of the called method.

Note: This example demonstrates the rules above for reference counting. We strongly recommend not to call retain() and release() manually. See the variant of this example in Handle class (page 20) which is simpler, obtain simpler, less error-prone and exception-safe.

2.10 Handle class

To simplify keeping track of reference counting (page 17) and necessary release() calls on interfaces (page 17), Iray offers a handle class, `mi::base::Handle`. This handle class maintains a pointer semantic while supporting reference counting for interface pointers. For example, the `->` operator acts on the underlying interface pointer. The destructor calls release() on the interface pointer, copy constructor and assignment operator take care of retaining and releasing the interface pointer as necessary. Note that it is also possible to use other handle class implementations, for example, `std::shared_ptr`[1] (or `boost::shared_ptr`[2]).

The handle class has two different constructors to deal with ownership of the interface pointer. See the `mi::base::Handle` documentation or the examples below for details.

It is also possible to use other handle class implementations, for example, `std_shared_ptr` (or `boost::shared_ptr`). In case you prefer to use such handle classes, you have to ensure that their destructor calls the `release()` method of the interface pointer. This can be achieved by passing an appropriate argument as second parameter, for example:

```
std::shared_ptr p (
    mi_neuray_factory(), std::mem_fun(&T::release));
```

2.10.1 Example 1

Assume there is an interface `IFoo` derived from `mi::base::IInterface`, and an API method that creates instances of `IFoo`. For example:

Listing 2.6

```
class IFactory : public mi::base::Interface_declare...>
{
public:
    virtual IFoo* create_foo() = 0;
};
```

Let `factory` be an instance of IFactory. As described by the rules in "Reference counting" (page 17), the implementation of create_foo() increments the reference count for the caller.

1. `https://en.cppreference.com/w/cpp/memory/shared_ptr`
2. `https://www.boost.org/doc/libs/1_68_0/libs/smart_ptr/doc/html/smart_ptr.html#shared_ptr`

Since the default constructor of `mi::base::Handle` leaves the reference count unchanged, you can simply use it to capture the return value. The destructor or the reset() method of `mi::base::Handle` decrement the reference count again via `mi::base::IInterface::release()`.

Listing 2.7

```
mi::base::Handle<IFoo> foo(factory->create_foo());
```

Using `mi::base::Handle` (or similar helper classes) instead of manually calling retain() and release() is strongly recommended for exception-safe code.

2.10.2 Example 2

Assume you want to implement the following interface.

Listing 2.8

```
class IRegistry : public mi::base::Interface_declare...>
{
public:
    virtual void register_foo(IFoo* foo) = 0;
};
```

Further assume that the implementation of `IRegistry` needs to reference the registered instance of `IFoo` after the method `register_foo()` returned.

Listing 2.9

```
class Registry_impl : public mi::base::Interface_implement<IRegistry>
{
public:
    Registry_impl(IFoo* foo);
    void register_foo(IFoo* foo);
private:
    mi::base::Handle<IFoo> m_foo;
};

Registry_impl::Registry_impl(IFoo* foo)
  : m_foo(foo, mi::base::DUP_INTERFACE) { }   Or: m_foo(make_handle_dup(foo))
void Registry_impl::register_foo(IFoo* foo)
{
    m_foo = make_handle_dup(foo);   Or: m_foo = mi::base::Handle<IFoo>(foo,
                                    mi::base::DUP_INTERFACE);
}
```

In this case you cannot use the default constructor of `mi::base::Handle` since you need to increment the reference count as well. This is done by the constructor that takes `mi::base::DUP_INTERFACE` as second argument. Alternatively, you can use the inline function `mi::base::make_handle_dup()` which does that for you.

Note that the default implementation of the copy constructor, assignment operator, and destructor of `mi::base::Handle` just do the right thing, and therefore, there is no need to implement them for Registry_impl. Also `NULL` handling for `foo` and `m_foo` is for free.

2.11 Resources

In general, you should aim for minimal resource usage. This implies releasing interface pointers as soon as you no longer need them. When using the `mi::base::Handle` class, it can be beneficial to introduce additional ... blocks to enforce the destruction of the handle, and release of the corresponding interface pointer, at the end of the block. Alternatively, you can also call `mi::base::Handle::reset`() or assign `0` to the handle.

2.12 Strings

There is an interface `mi::IString` representing strings. However, some methods return constant strings as a pointer to `const char` for simplicity. These strings are managed by the Iray API and you must not deallocate the memory pointed to by such a pointer. These pointers are valid as long as the interface from which the pointer was obtained is valid. Exceptions from this rule are documented with the corresponding method.

2.13 Error handling

Detecting failures depends on the particular method in question.

Some methods indicate their success or failure by an integral return value, for example, `INeuray::start`(), as already described in "Starting and shutting down the Iray API" (page 5). The general rule is that `0` indicates success, and all other values indicate failure.

Methods returning interface pointers indicate failure by a `NULL` pointer. Therefore you should check returned pointers for `NULL`. If you use the provided handle class, you can do so by calling `mi::base::Handle::is_valid_interface`().

Yet other methods report errors in more specialized ways, such as the importer methods on the `IImport_api` interface, which return a `IImport_result` object to handle sophisticated error reporting on file imports.

The program `example_configuration.cpp` (page 207) and subsequent examples use a helper macro called check_success() to check for errors. If the condition is not true, the macro prints an error message and exits.

> **Note:** For production software, thorough error-handling needs to be done. Detailed error-handling is omitted in the examples for simplicity.

2.14 Database, transactions and scopes

All Iray scene data is stored in the Iray scene database. The scene database is a distributed key-value store which allows to store scene elements and retrieve them later when they are needed. The keys are arbitrary strings, the value are instances of C++ classes. The fact that the database is distributed means that data can be stored on any node in a cluster and retrieved subsequently on any other node in the cluster.

The database is represented with the `IDatabase` interface in the API. It can be accessed through the `INeuray::get_api_component`() function.

Iray has been designed for multi-user operation and multiple operations running in parallel. This is supported in the database with the concepts of "Transactions" (page 23) and "Database scopes" (page 23).

You can create, retrieve and modify scene objects in the database, such as cameras, options, groups, instances, objects, meshes, and functions. Any thread which has access to an open transaction can at any time create and edit database objects. The database API is written to support multi-threaded accesses. It is the applications responsibility to synchronize concurrent changes to single database objects, though. It is also the applications responsibility that concurrent changes from different threads result in an overall consistent database content.

See also:

- "Scene database" (page 76) for the available scene database elements.
- "Creating a scene programmatically" (page 77) for an example of creating scene database elements through the Iray API.

2.14.1 Transactions

The use of transactions allows to run multiple operations in parallel. Imagine a rendering is running when a user starts to edit a scene and for example moves some geometry. If this change would immediately affect the ongoing render operation then the resulting image would be corrupt, because in some parts of the image the geometry would be still in the old position and in some parts it would be in the new position.

To avoid this the Iray database uses transactions to provide a consistent view on the database for the lifetime of a transaction. Before a rendering is started a new transaction is created and all accesses and changes to the database are done through that transaction. Changes from transactions which were not committed before the rendering transaction was started would not be visible to that transaction. In databases this is known as the *isolation* property.

Additionally it is possible to commit or abort a transaction. If a transaction is committed, all changes to the transaction become visible at the same time for all transactions that are created after the commit. If the transaction is aborted, all changes in the transaction are automatically abandoned and will not be visible for any other transactions. This is known as the *Atomicity* property.

Transactions fully work on distributed clustered systems, and each of the nodes in a system is able to start and commit transactions and use them for operations.

Note that the neuray database does not provide the *durability* property of conventional databases. All storage is done to main memory for fast operation. In the case of memory overusage, neuray can offload stored data to a disk. This is not meant to be done for persistent storage. Using external code, you can write data to conventional databases or plain files or read data from them.

Transactions are created from a scope (page 23), which in turn are created from the scene database (page 76). A transaction is represented with the `ITransaction` interface in the API.

2.14.2 Database scopes

Transactions (page 23) enable multiple operations to run at the same time without affecting each other. For multi-user operations a second concept is important: the *database scope*. A

scope is a container for database elements. Each database element is stored in a specific scope. Scopes can be nested, each scopes can have an arbitrary number of child scopes and an arbitrary number of database elements. Each scope except the “global” scope has exactly one parent scope. In that aspect the database is analogous to a file system with directories and files. A scope is analogous to a directory, while a database element corresponds to a file. The “global” scope corresponds to the “root” directory of a file system.

Scopes allow to have different database elements stored using the same key. This can for example be used to have different versions of the same database element for different users. An example where this is useful is, if two different users would like to view the same model of a car but have different colors on the paint. The usage of scopes is not limited to different users, they can also be used to have different variants of a model and let a user switch between them.

Each transaction is started within one specific scope. To carry the file system analogy further, the scope in which a transaction is started corresponds to the current working directory in a file system.

When some operation accesses a key for which multiple versions exist in different scopes the database needs to decide which version to return. This is done by first looking inside the current transaction’s scope. If the key is found there the database element stored in that scope is returned. Otherwise the scope’s parent scope is searched for the key. This continues until the “global” scope has been searched. If the key was not found in that scope the key does not exist or is not accessible from the current transaction’s scope.

The sketched mechanism allows to overwrite a shared version of some scene element individually for different scopes.

Scopes can be created and removed using API functions. Removing a scope will automatically remove all contained scene elements.

Scopes are created from the scene database (page 76). A scope is represented with the `IScope` interface in the API, which allows you to create transactions (page 23).

2.15 Runtime configuration

Iray does not use configuration files, command-line options or environment variables. Instead all configuration is done by calling configuration functions on the neuray object. Most configuration options need to be set before starting up the system but some can be adapted dynamically. See the following sections for details.

The following configuration sections provide a short overview of the available configuration options. See the corresponding `Configuration Interfaces` page for the complete set.

> **Note:** Note that “Logging configuration” (page 24) contains an example program to illustrate a configuration that most likely you want to do in your applications as well.

2.15.1 Logging configuration

The logging in neuray is done using an abstract C++ interface class with one member function which is called by neuray whenever some part of the system needs to emit a log message. You can provide a log object, which is an implementation of this abstract interface, and register it with the Iray API. Only one log object can be registered with a neuray process at any time.

In a multi-hosted system all log messages will be sent to an elected logging host. This host will pass log messages from all hosts to its log object. You can influence the log election process to favor a certain host.

Each log message will come with an associated log level and with a log category. You can decide which log messages to report to the user. The method of the log object that you have registered with the Iray API can be called at any time by any thread in the neuray system including application threads which initiated operations in neuray. The log object must be implemented to handle this concurrency properly.

You can configure a log verbosity level and neuray will pre-filter the log messages to avoid the processor load associated with generating log messages which are never presented to the user.

2.15.2 Example – Configuring the Iray API

Source code
`example_configuration.cpp` (page 207)

This example demonstrates how to configure the logging behavior of neuray. You can customize the logging behavior of neuray by providing your own logging object. This object has to implement the `mi::base::ILogger` interface. A very minimal implementation that prints all messages to `stderr` is shown in `example_configuration.cpp` (page 207). A similar implementation is used by default if you do not provide your own implementation.

2.15.3 Threading configuration

The neuray system internally uses a thread pool to distribute the load among the processor cores. The size of the thread pool is configurable. Additionally the number of threads which can be active at any time can be configured up to the licensed number of cores

The number of active threads can be changed dynamically at runtime. This can be used to balance the processor load against the embedding applications needs or other applications needs on the same host. If you decrease the number of threads it may take a while until this limitation will be enforced because running operations will not be aborted.

The application can use any number of threads to call into the Iray API. Those threads will exist in addition to the threads in the thread pool. The neuray scheduling system will however include those threads in the count of active threads and will limit the usage of threads from the thread pool accordingly.

2.15.4 Memory configuration

You can configure a memory limit. The neuray system will try to keep its memory usage below that limit. To achieve this it will flush data to disk. Objects can be flushed if the system guarantees that other hosts in the network still have them or if they are the result of a job execution which can be repeated.

neuray can be configured to use a swap directory to which it can flush out the contents of database element. Those elements can then be dropped from memory if the memory limit was exceeded. They will automatically be reloaded on demand when they are accessed again.

The memory limits can be adapted dynamically. If you decrease the memory limit neuray will make a best effort to reduce the memory usage to the given amount. Actually enforcing this limit may take a while and is not guaranteed to succeed.

You can configure neuray to use a custom allocator object provided by your application. The allocator has to implement an abstract C++ interface class which exposes the functionality of allocating and releasing memory blocks. It also allows one to inquire about the amount of memory currently being used. Calls to the allocator object can be done from several threads at the same time, including those threads from the application which are currently inside calls to neuray. The allocator object must be implemented to handle this concurrency properly.

If the allocator cannot provide the memory requested by neuray it needs to signal this failure by returning a 0 pointer. In that case neuray will try to flush memory to accommodate the request and retry again. If this is not successful neuray will terminate and release the memory it uses, but it cannot be guaranteed that all memory will be released. Additionally in that case it is not possible to restart neuray without restarting the complete process.

2.15.5 Rendering configuration

By default, rendering and related processes will use all elegible devices (CPU cores and GPUs) in the system. Which devices are eligible will depend on factors like hardware manufacturer, hardware generation, and driver.

Devices can be enabled or disabled per host or for an entire network cluster, per render mode or globally, via the `IRendering_configuration` interface. Hosts with and without enabled GPU rendering can freely be mixed in multi-hosted rendering. A device failure, e.g. lack of memory or a hard runtime error, or loss of a host from the cluster, will not necessarily cause rendering to stop. Renderers will generally attempt to continue rendering if at all possible, though the ability to do this depends on a number of factors. If CPU rendering is disabled (`IRendering_configuration::get_resource_enabled(IRendering_configuration::CPU,...)` indicates `false`) but CPU fallback is enabled (`IRendering_configuration::get_cpu_fallback_enabled()` returns `true`), rendering will continue on the CPU once all available GPUs have failed.

You can configure the location of the directories where scene files, textures, and MDL materials reside.

Rendering options which depend on the scene, such as trace depth etc., can be configured at any time and per rendered image by editing the options object in the database.

2.15.6 Networking configuration

Networking can be done in different ways: UDP multicast and TCP unicast with or without automatic host discovery are supported.

You can decide how to use the networking capabilities in your application. You can realize a conventional master/slave configuration similar to mental ray with satellites where all scene editing and rendering always initiates from the same host. In that scenario you would typically write a small stub application which does the configuration and starts the system and then waits until shutdown is requested. In the background the neuray system would handle reception of data and would accept requests to do work and handle them.

But you are not restricted to this scenario with neuray. The neuray library allows all hosts to edit the database and initiate rendering. You can use this to implement peer-to-peer

applications. It is up to your application to avoid conflicting changes in this scenario. To help doing that the Iray API provides means for synchronizing changes to database objects for example by locking objects.

UDP Multicast Networking: Using UDP multicast gives the best performance because data can be sent once on a host but be received by many hosts in parallel. Additionally there is no need to configure a host list so it is very easy to dynamically add and remove hosts. On the downside using UDP multicast for high bandwidth transmissions is not supported by all network switches and might require changes to the network infrastructure. For the UDP multicast case a multicast address, a network interface and a port can be configured. A host list is optional and acts as a filter which restricts the hosts which can join to the given list.

Hosts can be started dynamically and will automatically join without the need for configuration. A callback object can be given to the neuray system which will be called when hosts have been added to the network or when hosts have left the network.

2.15.6.1 TCP/IP networking

Because multicasting with high bandwidth is not supported on all networks it is also possible to use a more conventional scheme using TCP/IP networking, which is supported on virtually all networks. In that case an address and port to listen on can be configured. A host list is mandatory if the discovery mechanism is not used. Hosts can still be added to and removed from the host list at any time using the Iray API provided that the necessary redundancy level can be maintained (see below).

TCP/IP networking can be coupled with a host discovery mechanism, in which case an additional address needs to be given. In case of a multicast address, multicast will only be used to discover other hosts dynamically. In case of a unicast address, the host with the given address acts as master during the discovery phase. In both cases, the actual data transmission will be done using TCP/IP. Because this mode requires only low bandwidth multicasting it is supported by most networks and can be used to simplify the configuration even if high bandwidth multicast is not supported. Again a callback object can be given by the application to allow the application to keep track of added and leaving hosts.

2.15.6.2 Failure recovery

A redundancy level can be configured up to a certain maximum. The redundancy level configures how many copies of a certain database object will be kept at least on the network. The neuray database guarantees that the system will continue working without data loss even when hosts fail if the following preconditions are met: The number of hosts failing at the same time must be less than the configured redundancy level and at least one host must survive. After host failures or administrative removal of hosts the database will also reestablish the redundancy level if the number of surviving hosts is high enough

2.15.6.3 Dynamic scheduling configuration changes

A neuray instance in a multi-hosted system can dynamically be configured to stop delegating rendering work to other hosts, to stop accepting rendering work delegation from other hosts, or to exclude the local hosts from rendering completely. This can be used to adapt the load on systems to the current usage scenario

2.15.6.4 Administrative HTTP server

The neuray system has a built-in administrative HTTP server which can be started. This server is not identical to the HTTP server framework which can be used to serve requests from customers. The administrative HTTP server does not allow the execution of C++ code or the rendering of images. It is meant to be used to monitor the system at runtime. You can configure if this server is started (by default it is not started) and on which port and interface it listens. The administrative HTTP server allows one to inspect aspects of the neuray database and thus is useful for debugging integrations. Usually it would not be enabled in customer builds.

3 Rendering

The following sections describe general rendering concepts underlying the rendering implementation in Iray.

3.1 Rendering overview

The neuray rendering system supports GPU rendering on qualified hardware as well as software rendering using MDL materials. The available Iray render modes are "Iray Photoreal" (page 43) and "Iray Interactive" (page 62).

You can render from different application threads at the same time using the same or a different render mode and the same or different scenes or even different versions of the same scene at the same time.

To render an image you start with a render context that you obtain by calling the create_render_context() method of an `IScene` instance. The Scene object is a normal scene database element that you create and edit through a transaction. You initialize the scene by setting the root group, the camera instance, and the render options of the Scene object.

You can use the render context to render multiple frames. The rendered pixels will be written to a render target object, which is an implementation of an abstract C++ interface class. You can provide your application specific implementation of this interface class and pass it as a pointer to one of the render functions of the render context. This concept allows you for example to render to files, to render directly to the screen, or to render to textures. It also allows you to do progressive rendering. An example for a render target implementation is provided with the neuray release.

The render context is bound to one scene. You can use a render context to render from a single thread. The render call will return after the full image has been rendered or after rendering has been aborted. You can abort a rendering operation from a different thread using the render context object. However, aborting might not necessarily be instantaneous.

3.2 Rendering and canvases

Rendering is the operation of creating a 2D image from a 3D scene. The 3D scene is the data that is stored in the database.

For rendering, the scene data, in the form of a group node (the *root group*), needs to be combined with a camera instance and an options object into an IScene object. A camera instance and an options object are, for example, available in the `IImport_result` object returned by scene imports.

Rendering then happens in two steps; first you select a render mode by creating a render context from the scene, and second you call one of the render methods on the render context. The relevant methods are the following.

```
IRender_context* IScene::create_render_context(
    ITransaction* transaction,
    const char* renderer)
```

Returns a render context suitable for rendering the scene with a given render mode. See "Render mode selection" (page 32) for available render modes.

```
int32 IRender_context::render(
    ITransaction* transaction,
    IRender_target_base* render_target_base,
    IProgress_callback* progress_callback)
```

Renders the scene associated with this render context to the given render target according to the passed in parameters using the render mode associated with the context.

It is necessary to commit the transaction after calling this method before making any changes in the scene database that should be taken into account in a future render call. To the contrary, if no scene changes are necessary and future render calls are just used for progressive refinements, in that case it is beneficial for the rendering performance to use the same transactions for all progressive render calls.

This method returns when rendering is finished. The return value is non-negative if rendering was successful, and negative if an error occurred or the render was canceled. Progressive rendering returns one if the image is sufficiently refined according to the chosen quality parameters and returns the same image as in the last call in this case.

```
Sint32 IRender_context::render_async(
    ITransaction* transaction,
    IRender_target_base* render_target_base,
    IReady_callback* ready_callback,
    IProgress_callback* progress_callback)
```

Renders the scene associated with this render context to the given render target according to the passed in parameters using the render mode associated with the context.

It is necessary to commit the transaction after calling this method before making any changes in the scene database that should be taken into account in a future render call. To the contrary, if no scene changes are necessary and future render calls are just used for progressive refinements, in that case it is beneficial for the rendering performance to use the same transactions for all progressive render calls.

In contrast to the IRender_context::render call this call will return immediately while the rendering continues. The `ready_callback` is called when rendering is finished.

3.2.1 Progress reports

To obtain information on how a rendering is progressing, you can pass an instance of `IProgress_callback` to a render call. The callback's member function may be invoked at

various stages in the loading and rendering process and from various threads. Note that performing expensive operations in the callback thread may block rendering from progressing further.

The exact calls to this interface are subject to change without notice between releases and may depend on various combinations of options and other settings. Renderers will make a best effort to consistently provide the progress information in the following areas:

`iteration`
: Total number of samples rendered so far. The `value` parameter to the callback reflects the number of samples per pixel.

`canvas_update`
: Indicates that the application's `IRender_target` canvas was written. This may happen during progressive rendering, or only once all rendering has completed, depending on the current scheduling modes. In progressive rendering modes, it is safe to read the canvas during this callback, i.e., the renderer will not modify the canvas while reporting this progress area.

3.2.2 Abstract classes

The `IRender_target` class together with its related `ICanvas` and `ITile` classes are abstract interfaces which have to be implemented by the application which uses this API to render. This gives the application the ability to tailor the rendering process very specific to its needs. For a simple example implementation of these interfaces, see "A first image" (page 8). Also note that default implementations of `ICanvas` and `ICanvas_cuda` are available from the `IImage_api` interface.

The `IRender_target` class contains the canvases into which the render mode renders its results. The data written to a canvas is determined through the `IRender_target_base::get_canvas_type`() and `IRender_target_base::get_canvas_parameters`() member functions. The list of supported canvas types and parameters is documented in "Render target canvases" (page 182).

Given a rendering result in an `ICanvas` implementation, you can conveniently store it in an image format to disc or convert it to an image format in a memory buffer ready to be stored, encode it and give it to the video streaming server or send it as an answer to an HTTP request.

```
Sint32 IExport_api::export_canvas(
    const char* uri transaction,
    const ICanvas* canvas,
    Uint32 quality)
```

Exports a canvas, for example, to an image file on disk.

See "Scene file formats" (page 178) for the URI naming conventions supported for the uri parameter and "Image file formats" (page 181) for the available image formats and the quality parameter.

```
IBuffer* IImage_api::create_buffer_from_canvas(
    const ICanvas* canvas,
    const char* image_format,
```

```
    const char* pixel_type,
    const char* quality)
```

Converts a canvas into an image file format and returns it in a buffer. See "Image file formats" (page 181) for the available image formats and the quality parameter.

3.3 Render mode selection

The value assigned to the `render_mode` parameter of the `IScene::create_render_context()` method sets the render mode used to generate images.

Certain render modes are only available if their corresponding plugin is loaded. See the `IPlugin_configuration` interface for methods to load plugins.

The following table lists the possible string values that can be assigned to the renderer parameter, the required plugin, and the purpose of each render mode:

String value	*Plugin name*	*Purpose of render mode*
`irt`	`libirt`	"Iray Interactive" (page 62), a ray tracer with a focus on interactivity and editing. Iray Interactive is an interactive raytracing render mode which uses faster yet less accurate rendering algorithms than Iray Photoreal. Iray Interactive targets a look which is consistent with the physically based result of Iray Photoreal, yet it is optimized for interactive scene manipulations. Iray Interactive leverages NVIDIA CUDA-capable GPUs. It is ideal where ray-tracing effects, such as reflections and refraction, are desired and limited photorealism is acceptable.
`iray`	`libiray`	"Iray Photoreal" (page 43), a production-final renderer with full global-illumination support. Iray Photoreal is an interactive, ray-traced renderer that generates "push-button" photorealistic imagery featuring complex global illumination effects. Iray Photoreal makes use of a highly optimized BSDF and EDF shading framework, essentially simulating the physical behavior of real-world material and light sources. Iray Photoreal can leverage NVIDIA CUDA capable GPUs to render photorealistic images in a short amount of time. If a system does not have CUDA capable graphics cards it will automatically fall back to CPU based rendering, unless the `iray_cpu_support` restriction is set to false.

3.4 Progressive rendering of a scene

This section introduces:

- Progressive rendering (page 33), which explains what progressive rendering is and when to use it.
- `example_progressive_rendering.cpp` (page 265), which demonstrates how to set up progressive rendering for a render context.

3.4.1 Progressive rendering

Progressive rendering is intended to provide users with fast feedback. The initial render call generates a somewhat low quality frame. Subsequent render calls refine the image incrementally.

Each frame generated by a progressive rendering operation is exported to a file. Displaying all files in sequence shows the incremental refinement of the image. On Linux, you can view this progression using the `animate` command from ImageMagick software:

```
animate -delay 10 '\ls -1 example_progressive_rendering_*.png'
```

Several termination criteria determine when a progressive rendering operation is considered finished. The progressive rendering operation is restarted, for example, when the camera position changes or changes are made to the scene that would be visible in the rendered image. On the other hand, progressive rendering ends when the image resolution of the last rendered frame satisfies the image resolution criterion.

See Progressive rendering (page 49) in Iray Photoreal and Progressive rendering (page 65) in Iray Interactive for detailed information about progressive rendering options supported by the respective rendering modes.

3.4.2 Example – Progressively rendering a scene

Source code
`example_progressive_rendering.cpp` (page 265)

The example for progressive rendering is a modification of the example program used in "A first image" (page 8), "Accessing the database" (page 9), and "Importing a scene file" (page 9).

Note the following:

- The `interactive` scheduling mode (default mode) is used.
- The rendering logic has one modification: A loop is used to repeatedly call the render() method.
- A custom canvas class is used, whose implementation you can see in "Custom canvas class" (page 36).

3.5 Section objects

Objects in the scene may be sliced open by placing section objects in the scene. Section objects are modeled by the `ISection_object` interface. Section objects remove all objects on its front or outside from the scene. For example, placing a section plane at the origin with its normal pointing upwards will remove everything above zero altitude.

By default, the effect of introducing section objects is similar to removing clipped geometry from the scene entirely. This means that areas which were previously in shadow may now receive light because occluders were clipped away. This behavior may be controlled through the `ISection_object::set_clip_light` function.

If desired, sections may be closed with caps. If this mode is enabled, solid areas that were cut open by a section object are closed with a virtual surface. Note that the normal orientation will be used to determine which parts of an object to cap and that shading and geometry

normals must be oriented in a consistent fashion. As a consequence, nested objects will only be capped correctly if the outer object has a cavity (normals facing inwards) which then contains the inner object (normals facing outwards). Engineering models and geometry with a similar focus on fidelity are generally modeled in a suitable manner.

Also note that objects will not be capped correctly if the implicit environment ground plane runs through them.

The capping feature is controlled by the following attributes on the `IOptions` class:

`bool section_caps_enabled = false`
Enables capping of clipped geometry as described above.

`mi::Color section_caps_color = mi::Color(0.72f, 0.1f, 0.05f, 1.0f)`
Controls the diffuse color of the material that is applied to section caps. This color may be overridden per section plane via `ISection_object::set_cap_color`.

Another possibility to color the section of a cut object is to use subsurface scattering. Like caps this relies on the normal orientation to distinguish inner and outer parts. An object that already has subsurface scattering will look as if rays started directly inside the medium. However, you can also set volumetric attributes for otherwise solid materials. These attributes will never be used while the object is closed, but can be used to stir the look when cut open. To achieve a solid, colored appearance it is best to set a very large coefficient with an albedo of the desired color. This may also be used in combination with subsurface inhomogeneous volumes : (page 105).

3.6 Instancing

Iray supports two ways of handling scene geometry: flattening and instancing. Several modes are available to control how and which geometry is instanced. The choice of mode is controlled by the following attribute on the `IOptions` class:

`mi::IString iray_instancing`
Controls the instancing mode in Iray, which can be one of the following values, where `"off"` is the default:

`"off"`
If instancing is disabled (the default), all scene geometry is flattened into a single memory block. Geometry that is instanced multiple times in the scene will be duplicated. This mode almost always leads to higher memory use than using instancing `"on"`, but often yields higher rendering performance.

`"on"`
If instancing is enabled, all instanced geometry will only exist once, so input scene instances will also be instances in the rendering core. This may yield a significantly lower memory footprint, but may incur significant runtime costs, especially if the geometric extent of the instances overlap. Iray will also apply incremental object transform updates when instancing is enabled. This mode significantly reduces the scene preprocessing runtime overhead when moving objects around dynamically.

`"user"`
Without further intervention, this mode behaves like the `"off"` mode. This mode allows for fine-grained control over what is instanced and what is flattened. Scene elements like objects, groups, and instances can be tagged for instancing, as

explained in the following section. Iray will also apply incremental instance transform updates when user instancing is enabled. This mode significantly reduces the scene preprocessing runtime overhead when moving around (flattened) instances.

`"auto"`
If instancing is set to auto mode, Iray will automatically detect and decide which objects to instance, in order to reduce the memory footprint and speed up object transform updates. Input scene instances will usually be all instanced in the rendering core, unless there is a significant downside for memory or performance. This mode may significantly reduce the scene preprocessing runtime overhead when repeatedly changing the transformation of a group or (flattened) instance. In addition, this mode responds to the same controls as the user mode.

3.6.1 User control of instancing

In the user and auto modes of instancing, instancing can be controlled with the following attribute on the `IAttribute_set` class, which can be set on lights, objects, groups, and instances in the scene graph:

`bool movable = false`
If set to true, the respective node is marked for instancing.

If a group is marked for instancing, all of its contents will remain as one block, and that block will be instanced as per the scene graph. If the group contains multiple instances of some geometry, that geometry will be duplicated accordingly.

If an instance is marked for instancing, only this copy will be broken out. All remaining instances will still be flattened together. This is useful for cases where a few instances should be instanced but instancing all copies is not desirable.

It is generally most useful to mark groups for instancing. For example, consider a scene with two instances of a car. By default, all geometry (that is, two full cars) will be flattened into one big block. Marking the car group as movable will result in two instances of on car geometry (close to 50Further marking the left door group as movable will yield a scene with four instances and two geometry blobs: two instances each of a door and a car minus a door.

3.6.2 Increasing computational precision and accuracy

When instancing mode is set to “on”, “auto” or “user”, the scene is automatically shifted in space. This increases the precision and accuracy of rendering computations and reduces precision artifacts that originate from self-intersection. To disable this behavior, set `bool instancing_shift_camera` to “false”.

It is also possible to force an update of the scene shifting by calling the API method IScene::set_dirty at the beginning of a frame with a flag value that marks the instance transforms as dirty. This is recommended if the camera was moved a long distance (like when rendering new frames of an animation, or the user interactively changing it over time), as there is no automatic update of the shifting in order to avoid unexpected stutter through scene updates when moving the camera.

3.7 Custom canvas class

This section introduces:

- The header file `example_render_target_advanced.h` (page 279), which demonstrates how to set up a custom rendering infrastructure when the default rendering infrastructure is not sufficient for your needs.

3.7.1 Customized rendering infrastructure

Source code
`example_render_target_advanced.h` (page 279)

To render a scene, you have to provide some rendering infrastructure for the Iray API. Basically, you have to provide buffers where the rendered image is written to. These buffers are defined by the interfaces `IRender_target`, `ICanvas`, and `ITile`.

Most of the example programs shipped with Iray use internal default implementations. However, the example for progressive rendering (page 32) demonstrates an advanced way of providing the required rendering infrastructure by using custom implementations of the interfaces `ICanvas` and `ITile` and the corresponding classes `Canvas` and `Tile`. These advanced implementations are defined in this section's example source code.

For simplicity, the only supported pixel type is Color, that is, one `floatfloat` per channel.

You can see the canvas class in use in:

- Progressive rendering of a scene (page 32)

3.8 Asynchronous rendering

When using neuray, an application uses a render context for doing the actual rendering. A render context allows you to call the render() function. That call will initiate rendering and will not return before the rendering is done and an image was generated and stored for all requested render targets. You can give an optional callback function which reports about the progress of the rendering and may be called before the call to render() returns.

Especially for interactive applications, the blocking behavior of the render call can be inconvenient, because the application can not react to user events during the time of the rendering. For that reason the render context class offers an additional function named render_async(). When your application calls that function, the rendering will be initiated as in the render() function. But in contrast to that function, the call will return immediately without waiting for any result. Rendering will proceed in the background and your application can do other things during that time. To enable your application to find out, when the rendering is finished, you should give an additional callback function to the render_async() function callback, the ready_callback. That callback will be called when the rendering is finished.

Note, that it is important that you do not commit or abort the transaction as long as the asynchronous rendering is running. Otherwise, the rendering will be canceled immediately. In that case the result of the rendering so far would be discarded.

Note also, that neuray and all its renderers are fully multi-threading safe. That means as an alternative to using render_async(), you can create additional render threads in your application which then call the blocking render() function. Other application threads can still

react to user input and other events. The effect with this approach would be very similar to using the render_async() function.

3.9 Multiple render contexts

Iray has core support for storing multiple scenes at the same time. Your application can use this for example to render a main scene and in addition render swatches for the available materials and update those swatches whenever the materials change. Many other use cases for this feature are conceivable.

In addition to multiple scenes, Iray also supports multiple render modes, and in fact comes with different render modes featuring different quality and speed properties (like "Iray Photoreal" (page 43) and "Iray Interactive" (page 62)). As an application for this feature, you can for example choose one render mode for rendering during navigation and use a different render mode for rendering out a final image. It is also possible to blend between images generated by the different render modes.

To enable the efficient use of those features, Iray allows you to have many render contexts at the same time. Each render context can be used to render a specific scene with a specific render mode. You can switch very fast between scenes or render modes by using the render() function on one render context and when rendering is finished call render() on a second render context. In fact, it is also possible to call render() on a render context, while rendering is still active on other render contexts. Depending on the resource needs, the involved render modes, and the available resources, the rendering may in that case proceed in parallel, thus making more efficient use of the hardware resources.

3.10 Denoising

Iray Photoreal and Iray Interactive include two separate methods to reduce noise in generated images. One can select between a simple degraining filter and a Deep Learning-based denoising filter that operates as a post-process on the rendered image and uses additional data from the renderer to guide denoising. While the classic degraining filter can provide a moderate improvement in perceived image quality, the Deep Learning-based denoiser typically provides more drastic changes. While it is possible to enable both methods at the same time, this is not recommended and can lead to unsatisfactory results. The Deep Learning-based denoising filter operates on high dynamic range images. Tone mapping can be applied after denoising. TensorCores are used for inferencing on Volta and later GPUs.

Note:

- The Deep Learning-based denoising filter is only available when running on GPU hardware, since the inferencing step of Deep Learning systems is impractical to execute on CPUs. Also note that the additional memory required for denoising might exhaust device memory if almost all available memory is already in use by the renderer. In this case, the whole postprocessing system will fall back to CPU in order to preserve device memory, in consequence disabling denoising. Image resolution has a large impact on memory consumption of this denoiser.
- The Deep Learning-based denoising filter is intended for Iray Photoreal and Iray Interactive render modes only.

3.10.1 Simple degraining filter

The following attributes on the `IOptions` class, shown here with their default settings, control the simple degraining filter:

`mi::Sint32 iray_degrain_filtering = 0`

Selects one variant of the degraining filter. It can reduce low frequency noise without sacrificing overall sharpness. It is intended to be used in the final stage of the rendering phase, mainly to reduce remaining subtle grain in difficult areas of a scene. Setting this value to zero disables the degrain filter. There are five different modes to select from:

1. Pixel clipping
2. Smart median
3. Smart average
4. Limited blur
5. Limited auto blur

Modes 1 to 3 are working very conservatively and should thus be safe to use in general, modes 4 and 5 are considered to be more aggressive and should be used with caution, especially if the scene features fine details in either geometry or materials. It is recommended to experiment with different radius settings interactively to achieve best results.

`mi::Sint32 iray_degrain_filtering_radius = 3`

This value should be reduced if the filter blurs edges excessively and should be increased if some noise still remains in the image. This value is currently limited to a minimum radius of 2 and a maximum radius of 4.

`mi::Float32 iray_degrain_filtering_blur_difference = 0.05`

Modes 4 and 5 feature an additional setting that limits the influence of neighboring pixels if the brightness levels between are too different.

Note: Unfortunately there is no general recommendation on which degrain filter mode will perform best on a specific scene, thus it is necessary to cycle through the different modes manually to obtain the best results for a given scene. As a rule of thumb, modes 3 and 5 are usually the most reliable ones, with 5 being the more aggressive of the two.

Note: The following options are now deprecated:

- `progressive_rendering_filtering`
- `iray_black_pixel_filtering_max_frame`

3.10.2 Deep Learning-based Denoiser

The following attributes on the `IOptions` class, shown here with their default settings, control the integrated Deep Learning-based denoising:

`bool post_denoiser_available = false`

Makes the denoiser available to run. By itself, this option does not cause any changes in the output image (see option `post_denoiser_enabled` below), it just triggers generation

of all additional data needed for denoising to make sure it is available in case denoising is enabled later during progressive rendering. If set to true, this option may incur a small memory and performance overhead and should be set to false if denoising is not desired.

`bool post_denoiser_enabled = false`
If set, the denoiser will process the rendered image. If the option `post_denoiser_available` is not set, this option has no effect. It is not necessary to re-start progressive rendering of an image when changing this option, instead the next frame of the image will be produced with the selected value applied. Enabling the denoiser may consume significant memory and performance.

`mi::Sint32 post_denoiser_start_iteration = 8`
If the denoiser is enabled, this option will prevent denoising of the first few iterations, with the value specified in this option being the first denoised iteration. This can prevent the denoiser's performance overhead from impacting interactivity, for example, when moving the camera. Additionally, the first few iterations are often not suitable as input for the denoiser due to insufficient convergence, leading to unsatisfactory results.

`mi::Sint32 post_denoiser_max_memory = 1024`
Defines the maximum amount of device memory the denoiser is allowed to use in Megabytes (default: 1024 MiB). Note that the denoiser may automatically use less memory than the value specified here, depending on hardware used and current renderer memory usage, in order to balance denoising speed and memory usage. It should normally not be necessary to change this value, even on devices with very limited memory.

`bool post_denoiser_denoise_alpha = false`
If set to true, the alpha channel of RGBA images will be denoised, otherwise it will be left unchanged. Note that setting this to true will approximately double the time needed for denoising. Older versions of iray always denoised the alpha channel, if present.

Note: It is highly recommended to keep the "progressive_aux_canvas" scene option enabled to not disturb the DL denoiser by aliasing artifacts. Also note that while older Iray versions did recommend to not combine DL denoising with bloom, it is not an issue anymore.

The denoiser is also able to perform upscaling. See the Render target canvases (page 182) section for details.

3.11 Deep-learning-based SSIM predictor

Iray Photoreal and Iray Interactive provide a method for estimating how far away a partially converged image is from an imaginary converged image. This estimate might be used, for example, to decide when a Monte Carlo render is finished, or to estimate the time to render completion, or for adaptive sampling, or to determine whether denoising is necessary.

The SSIM predictor is trained by comparing partially converged images to fully converged images, and measuring their difference using the Structural Similarity Image Metric (SSIM). For more information about SSIM, see Structural similarity.[1]

1. `https://en.wikipedia.org/wiki/Structural_similarity`

Like the denoiser, this post-processing stage comes with a built-in pre-trained model that has many of the same caveats as the denoiser. Be sure to read and understand the limitations of the DLDenoiser post-processing stage. Be aware that the performance of the SSIM predictor model depends on the noise and color characteristics of the renderer used to product the source image. Also note that it is highly recommended to keep the "progressive_aux_canvas" scene option enabled to not disturb the process by aliasing artifacts.

3.11.1 Controlling the SSIM predictor

The following attributes on the `IOptions` class, shown here with their default settings, control Deep Learning-based SSIM prediction:

`bool post_ssim_available = false`
: Makes SSIM available. By itself, this option does not enable SSIM (see option `post_ssim_enabled` below), it just triggers generation of all additional data needed for SSIM to make sure it is available in case SSIM is enabled later during progressive rendering. If set to true, this option may incur a small memory and performance overhead and should be set to false if SSIM is not desired.

`bool post_ssim_enabled = false`
: If set, the SSIM predictor will process the rendered image. It is not necessary to re-start progressive rendering of an image when changing this option, instead the next frame of the image will be produced with the selected value applied. Enabling the SSIM predictor may consume significant memory and performance.

`float32 post_ssim_predict_target = 0.98`
: The SSIM target value for the prediction. The higher the value, the closer is targeted image to the fully converged one, with value of 1.0 zero being pixel-by-pixel match. Recommended range for this value is between 0.8 and 0.99. For values outside of that range, a warning is printed and prediction is computed for the target value clipped to that range.

`mi::Sint32 post_ssim_max_memory = 1024`
: Defines the maximum amount of GPU device memory the SSIM predictor is allowed to use in Megabytes (default: 1024 MiB). Note that the SSIM predictor may automatically use less memory than the value specified here, depending on hardware used and current renderer memory usage, in order to balance SSIM prediction speed and memory usage. It should normally not be necessary to change this value, even on devices with very limited memory.

3.11.2 Progress information

Depending on the active configuration, progress will be reported to the `IProgress_callback` in the following areas:

`progression`
: Estimated progress between 0 and 1 towards the configured prediction target.

`error`
: Estimated average per-pixel error.

`estimated_convergence_at_sample`

Reflects a very rough estimate of the total number of samples required to reach the configured prediction target.

`estimated_convergence_in`
Reflects a very rough estimate of the remaining amount of time required to reach the configured prediction target. This area is not available in all render modes.

Note that values reported here are rough estimates which will be affected by scheduling configuration and internal scheduling decisions, active hardware, etc.

3.11.3 SSIM predictor performance

For HD and 4K images, the SSIM predictor produced the following memory footprints and timings:

Size	*GPU*	*Memory*	*Time (ms)*
1920x1080	GV100	441 MB	8.7
	P6000	441 MB	9.2
	P100	444 MB	12.3
	P5000	444 MB	21
3840x2160	GV100	1765 MB	23
	P6000	1765 MB	28
	P100	1777 MB	44
	P5000	1777 MB	80

3.11.4 SSIM predictor limitations

The SSIM predictor has the following limitations:

- It is intended for Iray Photoreal and Iray Interactive render modes only.
- SSIM prediction is disabled if image denoising is set to enabled.
- It runs under the first GPU found by the system.
- There is no CPU fallback.

3.12 Matte fog

Iray Photoreal and Iray Interactive offer a simple "matte fog" effect to create fog-like atmospheric effects, such as aerial perspective, without the computational overhead of modelling an actual scattering volume. Matte fog does not affect rays hitting the environment, as attenuation and in-scattering are assumed to be already present in the environment or backplate. The following attributes on the `IOptions` class, shown here with their default settings, control matte fog:

`bool matte_fog = false`
Enables the matte fog effect.

`matte_fog_visibility = 10000.0`

Specifies the visibility range in meters, that is the distance at which the contrast between bright and dark objects is still perceivable according to the Koschmieder equation. Note that this value takes the conversion of scene to metric units into account (options attribute `"mdl_meters_per_scene_unit"`)

`matte_fog_visibility_tint = Color(1.0)`
Allows to vary the visibility range per color channel.

`matte_fog_brightness = 1.0`
Specifies the brightness of the in-scattered lighting, either as an absolute value or as a multiplier (see next item).

`matte_fog_brightness_relative_to_environment = true`
If set to true, the brightness of the in-scattered lighting is automatically determined to be relative to the total illuminance of the environment or sun and sky. If set to false, the brightness is an absolute value.

`matte_fog_brightness_tint = Color(1.0)`
Offers a color tint for in-scattered lighting.

4 Iray Photoreal

Iray Photoreal is an interactive, ray-tracing based render mode that generates "push-button" photorealistic imagery featuring complex global illumination effects. Iray Photoreal makes use of a highly optimized BSDF and EDF based framework, essentially capturing the physical behaviour of real-world materials and light sources. Iray Photoreal can leverage NVIDIA CUDA capable GPUs to render photorealistic images much faster than on a CPU, due to the highly parallel nature of the underlying simulation processes. If a system does not feature CUDA capable graphics boards it will automatically run on CPU. This will produce the same images but typically the rendering will take much longer to complete. In addition, it can also use both resources at the same time. See the `IRendering_configuration` interface and the related configuration section (page 26) for details.

The following sections provide an introduction to Iray Photoreal, document its system requirements and limitations, and provide further details about capabilities and options.

4.1 Introduction

Due to the simulation-based nature of Iray Photoreal, its behaviour is different to that of the other rendering modes in many aspects. For example most scene options which control global illumination and other advanced lighting aspects do not apply to Iray Photoreal.

Iray Photoreal does not support classic rendering features such as final gather or photon mapping but catches global illumination effects automatically without the need of explicitly controlling different stages of the rendering process and thus avoids tweaking various scene dependent parameters.

Due to this "push-button" simulation approach, it is not possible to exclude objects from casting or receiving global illumination, shadows, etc. The same is true for backface culling. Those flags will silently be ignored. It is also not possible to switch certain global illumination features off as these will always be included. The only exceptions are the effect of the `iray_max_path_length` rendering option that can for example be used to just get direct lighting contribution and the primary visibility flag for scene objects. In addition, the concept of light path expressions (page 190) was created to enable very similar workflows and effects instead, but in a much more general fashion that in addition is still physically plausible.

4.2 System requirements

The current system requirements for Iray Photoreal:

- The plugin `libiray` needs to be loaded in neuray before this render mode is available. See the `IPlugin_configuration` interface for methods to load plugins.
- Available on Windows x86 64 bit, Linux 64 bit, and macOS 64 bit.
- GPUs of CUDA compute capability 5.0 and higher are supported.
- Driver requirements: The driver needs to support CUDA 11.0.

4.3 Limitations

4.3.1 Result canvases

Iray Photoreal supports all render target canvases (page 182) with the following limitations:

- Iray Photoreal can currently render up to 20 light transport canvases at once. Light transport canvases are `result`, `irradiance`, and `irradiance_probe` canvases, including either built-in LPEs like `diffuse` or user-defined LPEs. Additional light transport canvases raise a warning and will be ignored. This limitation does not apply to auxiliary canvases like `object_id` or `normal`, which may still be computed in addition to this limit.
- Using objects with set `backplate_mesh` attribute in combination with the dome ground plane or other matte objects will result in undefined behavior, if an additional (non-backplate) object is intersecting the dome ground plane or the matte objects. In addition there is also a limitation where the ground plane, if seen from below, may still show the objects above the ground plane, even if the corresponding flag is set to false.
- Auxiliary canvases will yield unexpected results if the camera is placed inside a participating medium.
- Volume scene elements cannot be marked as "selected" for the `selection_outline` canvas.
- The number of texture spaces available through `texture_coordinate[` n `]` is currently limited to four.
- The `irradiance` canvas cannot be used in the same render call with any of the `result` render buffers.
- The `irradiance_probe` canvas cannot be used in the same render call with any other canvas.
- The `irradiance_probe` canvas cannot be used with the caustic sampler.
- Iray Photoreal cannot render the `shadow` canvas.
- Canvases held by `IRender_target_opengl` will always be written in their entirety, even if the `ICamera` specifies a window. Rendering will still be limited to the window area but the pixels outside of that area will be set to black.

4.3.2 Support of light path expressions

Iray Photoreal supports all light path expressions (page 190), with the following limitations:

- EDF specifications are not supported and will raise an error message.
- Directional lights are considered to be part of the environment for the purpose of LPEs. Note that, since the environment does not have an LPE label, directional lights cannot be filtered with labels.

4.3.3 Materials

The material model of Iray Photoreal is based on the Material Definition Language (MDL) (page 90). The implementation of MDL has the following limitations:

- The number of elemental BDSFs per material are currently restricted to a maximum of 16.

- The number of (procedural) texture functions per material are restricted to a maximum of currently 32.
- Spectral dispersion resulting from a spectral index of refraction (for example, specified by using base::abbe_number_ior) is currently only supported for df::specular_bsdf and not for df::simple_glossy_bsdf.
- The number of elemental VDFs per material is currently restriced to a maximum of 3. Further VDFs modify the directional bias of one of the first 3 VDFs by averaging.
- The material's volume coefficients are varying in MDL, however, Iray Photoreal only supports uniform coefficients.

4.3.4 Decals

Iray Photoreal has the following limitation on decals:

- The number of decals whose clip box can overlap at a surface shading point is limited to six. If a decal is applied on both sides of a surface, it naturally counts twice towards this limit. The number of decals in general is unlimited.

4.3.5 Rendering

Iray Photoreal has the following general limitation:

- Infinite lights are always considered matte lights.
- Iray Photoreal ignores the following attributes:
  - `reflection_cast`
  - `reflection_recv`
  - `refraction_cast`
  - `refraction_recv`
  - `shadow_cast`
  - `shadow_recv`
  - `face_front`
  - `face_back`

4.3.6 Picking

Picking can only be used on a render context that has already been used at least once to render. Otherwise picking will return with a proper diagnostic and not pick any object.

4.4 Matte objects and matte lights

4.4.1 Introduction

Matte objects and matte lights are advanced techniques to aid including synthetic objects under matching lighting (captured for example via lightprobes) onto a real world photograph (a virtual backplate (page 131) or environment map). In a classic workflow, one needs to assign special custom shaders to objects to receive a separate shadow pass along with

additional global illumination information that must then be combined with the original rendering pass in external compositing steps.

Iray Photoreal simplifies this classic workflow by avoiding the separate postprocessing steps and allowing for a fully progressive and physically plausible rendering of the complete scene at once. In addition, matte objects do not need to rely on secondary buffers and setting up the additional shaders that must usually be tweaked on a per scene basis, as all of the necessary logic is directly incorporated into the actual core rendering architecture.

To guide the interaction of backplate and rendered objects though, additional matte objects have to be provided that should (roughly) match the photographed real world objects using the same camera settings. In addition, these objects should be assigned (roughly) matching materials to further perfect the illusion of the rendered objects being part of the environment and especially interacting closely with the backplate.

4.4.2 Attributes

The following attributes on the `IAttribute_set` class controls matte objects:

`bool matte = false`
: If set to true, this attribute flags objects and lights as matte objects or matte lights, respectively.

`mi::Float32 matte_shadow_intensity = 1.0`
: If the matte flag is set for an object, this controls the intensity of the shadow cast on the matte object with a value greater or equal to 0.0. A value of 0.0 denotes no shadow at all and a value of 1.0 denotes regular shadow intensity, thus values between 0.0 and 1.0 denote a lighter shadow. Values above 1.0 increase the shadow intensity. Note that the range of useful values above 1.0 is rather limited and exceeding this range may lead to an unnatural dark shadow that can influence other additive lighting effects, such as reflections of the objects in the scene. The exact useful range is scene dependent.

`bool matte_connect_to_environment = false`
: Only effective if a backplate function or the backplate color is set. When set to true, secondary interactions (like a reflection) with the matte object will use the environment instead of the backplate/color.

`bool matte_connect_from_camera = false`
: This additional flag makes the matte object behavior toggle between two different lookup types. The first one is done locally (flag is set to true), where similar interactions with the matte object (for example, the same point of intersection) will deliver the exact same lookup into the real world photograph, which usually works best for a very elaborate matte object setup specifically arranged for only a single or very similar camera position(s). The second type is global (flag is set to `false`) where the real interactions with the environment or backplate are used instead, which is usually preferrable for simple matte object setups (like plain shadow catchers) or full camera interaction. Note that setting the flag will only be noticeable in scenes where no backplate/color at all is specified (so all interactions will always be happening with the environment), or if a backplate/color is specified and the `"matte_connect_to_environment"` flag is set in addition.

4.4.3 Scene Option

`bool matte_remap_uv_to_dome_in_aux_canvas = false`

If set to true, all matte objects (including the implicit dome ground plane) will feature special texture coordinates that are written into the texture_coordinate canvas. These can be used for the actual uv-mapping into a spherical environment texture map.

`bool matte_visible_in_aux_canvas = false`

When set to true, all matte objects (including the implicit dome ground plane) will show up in any auxiliary render canvas like texture_coordinate, object_id or distance.

`bool matte_area_lights_visible = false`

By default matte area lights are not visible (neither directly nor in reflections) since in a matte workflow those paths should evaluate to the backplate or the environment. This behavior can be changed by setting this option to true.

4.4.4 Example

A simple example would be to add a synthetic cube onto a backplate of a photographed wooden table.

- The scene must be set up to include both the cube and the top of the table (which can be a very rough approximation, so a simple rectangle the size of the table can already be sufficient).
- The virtual camera must be set up to match the original real world camera position and lens settings (to have the objects in scene and photograph align).
- An environment map with similar lighting characteristics (best would be a matching lightprobe of the same real world scene) must be provided.
- Approximately match the material of the original table, including textures (or even a bump or normal map) and enabling the matte flag of the table object.

Instead of rendering the table object directly into the output buffer like any other standard object, Iray Photoreal now triggers a specialized handling of the matte object that directly combines backplate information, environment lighting, interactions of both matte and synthetic objects (global illumination) and the artificial shadows cast onto the matte objects, in a single progressive rendering step.

4.4.5 Additional notes and limitations

An environment map or a backplate (or both) should be specified to get the full benefit of matte objects. These define the appearance of matte objects when they interact (through reflections and refractions) with synthetic objects. If `"matte_connect_from_camera"` is enabled, the backplate is always projected onto the matte object from the camera view. If this projection is impossible, the environment map is used instead. If a hemispherical environment dome of sufficient resolution and quality is available, this can also be used alone. But these are difficult to generate, so in most cases a 2D photograph will be used as the backplate, which can be seen as a high-quality complement to the environment map.

It is possible to use the matte objects feature with neither a backplate nor an environment map. In such a case, matte lights should be used to approximate the scene lighting. But many interesting effects will then be missing from the rendered image, such as reflections of matte

objects onto synthetic objects, because the backplate/environment defines how matte objects appear in those reflections.

Note: The implicit groundplane (page 119) is basically a special built-in case of the more general matte object functionality.

Note: When rendering irradiance (either probes or buffer) all matte objects are completely ignored and act as if the geometry is flagged as disabled.

4.5 Ghost lights

Area light sources in Iray Photoreal are created via regular scene geometry. As such, these are typically always visible, both directly (from the camera) and indirectly, meaning in reflection or refraction, and do cast shadows. This may in some situations be undesirable, in particular if light sources are placed by artists to create a certain look or lighting mood: the intention may then be that lights should illuminate the scene (to a large extent), but also should otherwise not show up directly in the rendering. This creates a new type of problem for physically based rendering though: When is the contribution of a light source considered to be wanted, and when should it be omitted, without jeopardizing the rest of the light transport simulation?

To get started, the basic building blocks that control the visibility of lights are:

- For camera rays the `visible` attribute may be used to exclude directly visible geometry.
- A transparent MDL material (thin-walled, with only a specular transmission BSDF) can be used to make the non-emissive part of the light geometry disappear, including shadows.
- Light path expressions can be used to classify and (exclude) certain light contribution paths from the rendering.

Iray Photoreal further adds the concept of *ghost lights* to complement these basic concepts. Ghost lights allow for more fine-grained control on how light sources are visible in glossy interactions. Such lights feature an additional factor to enable a smooth blend between full and no contribution in glossy reflection and transmission. The factor can be set as an attribute on the `IAttribute_set` class and may affect both objects emitting and objects receiving light:

`mi::Float32 ghostlight_factor = 1.0`

Values greater than one turn an area source of light into a ghost light. This will immediatelly remove contributions of that light source from paths reaching the light via pure specular reflection and transmission. The larger the value, the more of the emission from that source will be removed from glossy interactions, i.e. glossy highlights and reflections from that source will smoothly be faded out. Increasing the factor by one will halve the peak brightness of glossy highlights. Thereby, surfaces with higher roughness are influenced less. In any case, diffuse materials are excluded from the factor and will always be lit from the light source the same way. Note that this factor can also be set on the receiving object side, where it acts as a multiplier to the light source factor (but only if there are ghost lights in the scene to begin with). This allows to fine-tune the behavior on a per object basis to achieve both a stronger removal of highlights or their recovery by using factors less than one.

4.6 Refractive objects and IOR handling

4.6.1 Modeling Complex Refractive Objects

When building scenes with multiple refractive objects in it, a common way to deal with hierarchical volumes (that feature a varying index of refraction (IOR)) is to specify separate materials for the inside and outside of an object to provide a necessary hint to the rendering implementation about the order of the IORs. Another common trick to guide the rendering is to leave a small gap of "air" between the volumes to avoid numerical precision issues. In Iray Photoreal, these workarounds are not needed. It will handle all IOR stacking automatically and does not need to care about explicit surface orientation or precision.

Still, there can be different approaches on how to represent hierarchical refractive volumes. As an example, there are different ways to model a glass full of water. First of all we will assume that the glass itself will be modeled double-sided (solid), as this provides the most realistic look by definition. As for the water volume inside the glass, there are now three possibilities: Matching the boundary geometry of the inner glass layer exactly with the water volume geometry, or leaving a small gap of "air" in-between the two volumes, or making the water volume slightly overlap the glass volume.

While Iray Photoreal can handle all three cases automatically, the most realistic rendering can only be guaranteed by providing slightly overlapping volumes, due to the inherent limited numerical precision of the simulation process. The overlapping volumes will be detected and the simulation will handle the refraction calculations as if the volumes would perfectly align.

More simplistic cases, for example, ice cubes or bubbles floating inside the glass of water, do not need special care at all, and can be modeled in a straight-forward way and placed completely inside the other refracting object volume.

4.7 Progressive rendering

4.7.1 Progressive rendering modes, updates and termination

Iray Photoreal can render progressively into a render target. On each render call the image will be refined more so that over time the image converges to very good quality. The intermediate images can be displayed to the user thus giving a good early impression about the final lighting and material look of the scene. Note that Iray Photoreal does not use any final gathering or photon map based techniques so any changes to the lighting will be visible very fast. Progressive rendering works only if the same render context is reused for subsequent images.

There are a number of termination criteria that control when an image is considered completed. Unless an error is encountered, `IRender_context::render`() will return 0 until the image is completed, at which point the call returns 1.

For each render request, the renderer will automatically check if the parameters for the requested image (for example image resolution or camera position) fit to the last rendered image. Also the scene will be checked for changes. If any changes are found that would result in visible changes in the image the render context will be reset and the progressive rendering starts from the beginning.

The following key-value pairs, shown here with their default settings, control rendering when passed to `IRender_context::set_option`().

`const char* scheduler_mode = "interactive"`
: Controls the rendering mode, which is by default interactive. Setting this option to `"batch"` switches to the non-interactive rendering mode. This mode yields higher efficiency at the cost of responsiveness and is thus better suited to the generation of a final frame, rather than interactive display.

`mi::Uint32 min_samples_per_update = 1`
: Controls the minimum number of samples that need to be rendered per render call in the progressive render loop.

`mi::Float32 interactive_update_interval = 1.f/60.f`
: The update interval time in seconds is compared to the rendering time since the beginning of this interactive render call. This acts as a hint to the renderer. Longer times will generally increase rendering efficiency, while shorter times yield more frequent updates.

`mi::Float32 batch_update_interval = 1e36f`
: The update interval time in seconds is compared to the rendering time since the beginning of this batch render call. This acts as a hint to the renderer. Longer times will generally increase rendering efficiency, while shorter times yield more frequent updates.

`mi::Float32 max_progressive_update_interval = 300.f`
: A soft upper limit to the time between progressive updates. The internal scheduling mechanism will gradually transition from greater responsiveness to greater efficiency. This option limits that transition. This option can be used to ensure a certain level of responsiveness (e.g. time required to react to scene changes), even at later stages of progressive rendering, at the cost of rendering efficiency.

The update interval options control how frequently the render call returns to the caller. At the default settings, `IRender_context::render()` will return after each update in interactive mode, and after finishing the image in batch mode. Decreasing the `"batch_update_interval"` will allow the call to return before completing the image. The image is considered completed when either the maximum number of samples or the maximum time is exceeded, or if the error falls below a given threshold (see below), but not before the minimum number of samples have been computed.

All these termination criteria are checked after each progression step. The size of the steps is controlled internally and increased over time to improve efficiency when quick updates would no longer add significant information to the resulting image.

The following attributes on the `IOptions` class, shown here with their default settings, control progressive rendering:

`mi::Sint32 progressive_rendering_min_samples = 4`
: Controls the minimum number of samples that need to be rendered before the progressive render loop is allowed to terminate by any of the termination criteria.

`mi::Sint32 progressive_rendering_max_samples = 100`
: The maximum number of samples is compared to the number of samples since the beginning of this progression. The render call will return 1 if the rendering loop is terminated by this termination criterion.

  A value of -1 will disable the criterion altogether.

`mi::Sint32 progressive_rendering_max_time = 3600`
The maximum time in seconds is compared to the rendering time since the beginning of this progression. The render call will return 1 if the rendering loop is terminated by this termination criterion.

A value of -1 will disable the criterion altogether.

`bool progressive_aux_canvas = true`
Controls progressive rendering of auxiliary canvases like `depth` and `object_id`. If enabled, rendering auxiliary canvases will continue until one of the stopping criteria is met, thus potentially sampling each pixel with more than one sample. Note that the convergence quality estimate described below is still only computed for light transport canvases. However, rendering will be stopped for all canvases when the quality criterion is reached.

Note that it is highly recommended to keep the option enabled when using the AI Denoiser, DL based SSIM quality prediction, or Toon postprocessing, as otherwise the aliasing can disturb these filters and lead to noticeable quality loss, especially at a large number of iterations rendered.

If this option is disabled, auxiliary canvases only receive a single sample per pixel. This can be desirable sometimes because of the nature of the canvases: averaged normals or depth values are not always well defined (e.g. on silhouettes).

The non-interactive mode can be used to reduce overhead and thus to increase scalability for large render tasks, since the result is written to the canvas (and potentially transferred over the network) less frequently.

When the scheduler mode is switched from interactive to batch, Iray Photoreal will continue refining the image in batch mode at the next render call. When switching from batch mode to interactive mode, all progress will be lost and rendering will restart from scratch.

4.7.2 Convergence quality estimate

At some point the progressive rendering will have reached a quality where subsequent frames will not have any visible effect on the rendering quality anymore. At this point rendering may stop. This can be based on a convergence quality estimate that can be computed in the non-interactive rendering mode.

The following attributes on the `IOptions` class, shown here with their default settings, control the convergence quality estimate:

`bool progressive_rendering_quality_enabled = true`
The convergence quality estimate is only available in the batch scheduling mode and can be enabled and disabled with this attribute. If disabled, rendering will not stop based on the convergence quality and no progress messages will be issued for the current convergence quality.

`bool progressive_rendering_quality_ssim = false`
Switches the convergence quality estimate to be based on the SSIM predictor (page 39). Rendering will stop once the configured `post_ssim_predict_target` is reached. Note that the predictor must be enabled for this option to work. Also note that `progressive_rendering_quality_enabled` must be `true` for this option to work. Contrary to the standard quality estimate, SSIM-based termination is not limited to batch scheduling.

`mi::Float32 progressive_rendering_quality = 1`
A convergence estimate for a pixel has to reach a certain threshold before a pixel is considered converged. This attribute is a relative quality factor for this threshold. A higher quality setting asks for better converged pixels, which means a longer rendering time. Render times will change roughly linearly with the given value, so that doubling the quality roughly doubles the render time.

This option has no effect in SSIM mode.

`mi::Float32 progressive_rendering_converged_pixel_ratio = 0.95`
If the progressive rendering quality is enabled, this attribute specifies a threshold that controls the stopping criterion for progressive rendering. As soon as the ratio of converged pixels of the entire image is above this given threshold, Iray Photoreal returns the final result for forthcoming render requests. Additionally, the render call will return 1 in this case indicating to the application that further render calls will have no more effect. Note that setting this attribute to a value larger than the default of 0.95 can lead to extremely long render times.

This option has no effect in SSIM mode.

The convergence quality estimate has some computational cost. It is only computed after some initial number of samples to ensure a reasonably meaningful estimate. Furthermore, the estimate is only updated from time to time.

4.7.3 Deprecated attributes

`mi::Float32 progressive_rendering_error_threshold = 0.05`
Use `"progressive_rendering_quality"`, `"progressive_rendering_quality_enabled"`, and `"progressive_rendering_converged_pixel_ratio"` instead. In particular, `"progressive_rendering_converged_pixel_ratio"` can be set to one minus this value for the same effect.

4.8 Spectral rendering

Iray Photoreal supports spectral rendering, where light transport is not merely simulated on tristimulus color values but on a continuous range of wavelengths. Iray Photoreal simulates light on the spectrum of human visible color (380 to 780 nanometers, as defined by CIE color matching functions) and computes a color result. The conversion to color implicitly transforms the result from radiometric to photometric units.

While it is recommended to use spectral input data, it is of course possible to also use color input data (such as RGB color textures), if spectral data is not available. The conversion to spectra is automatically handled by the rendering core at runtime and there typically is no memory overhead.

Spectral rendering can be activated and controlled via a set of scene options.

`bool iray_spectral_rendering = false`
Enables spectral rendering.

`string iray_spectral_conversion_color_space = "rec709"`

For the conversion of color data to spectra, the rendering core needs to know the color space the data is defined in. Supported color spaces are CIE XYZ (`"xyz"`) Rec.709/linear sRGB (`"rec709"`) Rec.2020 (`"rec2020"`) ACES2065-1 (`"aces"`) and ACEScg (`"acescg"`)

> **Note:** This option also comes into play when spectral data is used without spectral rendering enabled: There it defines the conversion of spectra to colors to be used for rendering.

`string iray_spectral_conversion_intent = "faithful"`

If color data is used in spectral rendering it needs to be converted to spectral data. This conversion is ambiguous in the sense that for a given color there typically are infinitely many spectra that yield that color (metamers). Fundamentally, the conversion in Iray Photoreal is driven by the goal of yielding smooth spectra with some flexibility to steer the details for reflection colors. Valid options are:

`"natural"`
: Smoothness is preferred over reflectivity. As input colors approach the edge of the gamut, the intensity of the resulting spectrum necessarily decreases. Consequently, highly reflective saturated colors may render darker when spectral rendering is enabled.

`"faithful"`
: Sacrifice some smoothness to yield greater compatibility with color rendering

Note that this option only has an effect if `iray_spectral_conversion_color_space` is `"rec709"`. For other spaces, the behavior is similar to `"natural"`.

```
mi::Float32_3_3 iray_spectral_output_transform =
  mi::Float32_3_3(
    3.240600, -1.537200, -0.498600,
    -0.968900, 1.875800, 0.041500,
    0.055700, -0.204000, 1.057000)
```

This transform will be applied to the color result and can be used to transform to a desired result color space. The default transforms from CIE XYZ to Rec.709/linear sRGB.

`string iray_spectral_observer = cie1931`

The observer function maps spectral values to three channel color output. By default the photometric rendering mode uses the CIE 1931 2 degree standard observer as color matching functions, such that the color ouput is CIE XYZ. Setting this option to `"cie1964"`, the color matching functions can be changed to the CIE 1964 10 degree standard observer. Further, this option can be set to `"custom"` to use the user-defined observer curve provided with `iray_spectral_observer_custom_curve`.

`mi::Float32_3[] iray_spectral_observer_custom_curve = []`

Specifies a custom response curve as observer function. The values are equally spaced between the minimum and maximum wavelength as provided by `iray_spectral_observer_custom_wavelength_min` and `iray_spectral_observer_custom_wavelength_max`. Note that spectral rendering in Iray produces spectral radiance values in radiometric units (Watt per steradian per square meter per nanometer) and the observer curve then defines the how that relates to color output (in particular magnitude). For reference, the CIE color matching curves have a peak of 683.002 and produce luminance output in photometric units (Candela per square meter).

`mi::Float32 iray_spectral_observer_custom_wavelength_min = 400.0`
Specifies the minumum wavelength for the custom observer curve (in nanometers).

`mi::Float32 iray_spectral_observer_custom_wavelength_max = 700.0`
Specifies the maximum wavelength for the custom observer curve (in nanometers).

4.9 Caustic sampler

The Iray Photoreal caustic sampler can be used to improve the quality of caustics in typical turntable scenes, but also for scenes with complicated lighting and material setups. When enabled, Iray Photoreal will augment the default sampler with a dedicated caustic sampler, designed to improve capturing caustic effects. In terms of light path expressions (page 190), the caustic sampler was designed to handle paths with the signature `ED.*SL` much better.

The following attribute on the `IOptions` class, shown here with its default setting, controls the caustic sampler:

`bool iray_caustic_sampler = false`
Enables the caustic sampler if set to true.

The Iray Photoreal caustic sampler was mainly designed for typical turntable scenes, but is of course not limited to them, as improvements in rendering can be found with any moderately complex scene. Still, it will perform best under the following conditions:

- The scene geometry is (mostly) in view.
- The caustics are (mostly) in view.
- The scene is positioned on top of a groundplane or large surface.

> **Note:**
>
> - To catch caustics on the groundplane, the groundplane's `environment_dome_ground_reflectivity` parameter must be set. See "Environment dome" (page 119) for a description of the groundplane and its settings.
> - The caustic sampler only improves on directly visible caustics. Caustics seen through mirrors or windows are currently not improved by the caustic sampler.

The caustic sampler will not improve caustics in all use cases, and thus can actually harm overall rendering speed as it comes with some performance overhead. The caustic sampler can fail to improve caustics under the following conditions:

- Only a small part of a larger scene is in view (for example, the camera zooms in on a single item in a detailed scene).
- The camera zooms in on a small part of a much larger caustic.

4.10 Guided sampling

The Iray Photoreal automatic guided sampling can be used to improve both the quality of rendering (especially if the caustic sampler is disabled and/or the firefly filter enabled), as well as the rendering convergence speed (when rendering a complicated scene, such as large

interiors). When enabled, Iray Photoreal will augment the default sampler with a dedicated guidance cache, designed to improve the convergence of complicated light transport scenarios.

The following attribute on the `IOptions` class, shown here with its default setting, controls the improved sampling scheme:

`bool guided_sampling = false`
Enables automatic guided sampling if set to true.

The guided sampler will not improve convergence speed in all use cases, and thus can actually harm overall rendering performance as it comes with some performance overhead, especially on multi GPU systems, and even more when using hybrid GPU/CPU rendering. It also comes at the cost of an increased memory usage (roughly 100 MB on each device, may increase in future releases) and a slightly reduced iteration throughput, typically below 10 percent. The actual benefit can vary a lot, depending on the used hardware, if batch or interactive scheduling is employed, and the type of scene being rendered. Turntable-like scenes will not profit as much as architectural scenes that feature complicated lighting and materials. In general, the number of iterations per time budget will decrease, while image quality increases for the same budget. Scenarios, that are typically improved, are:

- Strongly occluded light sources. This includes bright environment map content (e.g. the sun) that is blocked by walls, but extends to all other strongly shadowed lights as well.
- Volumetric rendering.
- Soft shadows of large area lights. To work, this requires the area light to be tessellated.
- Moderate caustics. Note that very sharp caustics can still only be sampled with the caustic sampler (page 54).

This feature depends on the automatic discovery of sources of variance. In some situations this can also introduce more high frequency noise (speckles) due to some missed edge cases in training. Thus, (for now) the recommendation is to enable the firefly filter in combination with guided sampling.

4.11 Motion vectors

Motion vectors represent the direction and the amount of movement of the hit point of any pixel in a scene. Note the following:

- Motion vectors are an exact measure of the difference between the position of hit points at shutter open time and at shutter close time.
- Only first hits from the camera are considered. There is no record of the movement of a hit point for a secondary ray.

Motion vectors can be output to a separate buffer. The computation and the output of motion vectors is controlled by the following attributes of the `IOptions` class:

`bool motion_vectors_enabled = false`
When set to `false` (default), the output in the motion vectors buffer is black.

When set to `true`, the computation of motion vectors is enabled.

`mi::IString motion_vectors_space = "screen"`
Computes motion vectors in `"screen"`, `"camera"`, `"world"`, or `"NDC"` space.

By default, motion vectors are computed in screen space; they represent the difference in the number of pixels between shutter open time and shutter close time. In screen space, the Z-component of motion vectors is always zero, which means the resulting motion vectors are limited to two dimensions.

`mi::Float32_3 motion_vectors_scale = mi::Float32_3(1,1,1)`
Scales the motion vectors' components by the given amounts. For example, to flip the X or Y component of the vectors, set the scale factors to (-1,1,1) or (1,-1,1) respectively.

`mi::Float32 motion_vectors_shutter_offset = 0`
The value range is `0` to `1`. When set to a value greater than zero, the shutter open time is shifted. Shifting the shutter open time can be useful, for example, in 2D compositing tools due to the way motion vectors can be used. Shifting the shutter open time also affects the beauty buffer.

`bool motion_vectors_instantaneous_shutter = false`
When set to `"false"` (default), the beauty buffer contains a motion-blurred image.

When set to `"true"`, the shutter is frozen at the open time, which produces a beauty buffer without motion blur (motion vectors must be enabled for this to happen). This is very useful when producing motion vectors for compositing, when a non-blurred image is needed along with the corresponding motion vectors.

`bool motion_vectors_background = false`
When set to `"false"` (default), motion vectors are not computed for environment hits and the corresponding pixels in the motion vectors buffer are black.

When set to `"true"`, motion vectors are computed for environment hits as well.

4.12 Rendering options

One of the main design goals of Iray Photoreal is ease of use. It is not necessary to control the algorithmic behavior via a myriad of settings of this render mode to achieve good results across different kind of scenes.

Nonetheless, the following few attributes on the `IOptions` class, shown here with their default settings where applicable, can control some aspects of Iray Photoreal:

`mi::Sint32 iray_max_path_length`
Bounds the maximum number of segments (bounces) of light paths to contribute to the result. A length of two corresponds to direct light. Since this setting cuts off indirect lighting contributions (one example would be the headlight of a car that depends on a lot of indirect effects to look correct), it should only be applied when the rendering has to be accelerated at the expense of physical accuracy.

`mi::Sint32 iray_max_sss_path_length`
Bounds the number of volume (aka subsurface scattering) vertices on a path. Consecutive volume vertices are counted as a single event in the `iray_max_path_length` restriction. This allows to have more volume bounces than surface bounces since they tend to have larger contributions over long paths.

For example a path `ETV+TL` has a path length of four, regardless of the number of volume vertices which cannot be more than `iray_max_sss_path_length`. Note that single volume events count like regular events. A path `ETVRVTL` has a path length of six with two volume vertices, so either `iray_max_path_length<6` or

`iray_max_sss_path_length<2` would cancel this path. For details on the path descriptions see light path expressions (page 190)

`mi::IString iray_default_alpha_lpe`
Controls the default light path expression used for alpha channels if none is specified explicitly. The default is `ET*[LmLe]`. See "Canvas names" (page 188) for details.

`mi::Sint32 filter = FILTER_BOX`
Iray Photoreal natively supports `FILTER_BOX`, `FILTER_TRIANGLE` and `FILTER_GAUSS` filters for antialiasing. If `FILTER_CMITCHELL` or `FILTER_CLANCZOS` is requested, Iray Photoreal will use the default box filter instead.

`mi::Float32 radius`
The radius of the filter kernel. Recommended values are 0.5 for box filter, 1 for triangle filter and 1.5 for Gauss filter.

`bool iray_firefly_filter = true`
Controls a built-in filter to reduce bright spots that may occur under some difficult lighting conditions. Such bright undesired pixels are often called "fireflies".

The filter works best in combination with the built-in tonemappers. If a tonemapper is not enabled, the filter estimates can be too pessimistic for some scenes, resulting in some fireflies to still appear. Contrary, it can also happen that the filter cuts off noticeable contributions to the image, especially when calculating an irradiance buffer or irradiance probes, as in that case no internal tonemapping is enabled. So it is required to either disable the firefly filter completely or to set a correct `"iray_nominal_luminance"` value whenever there is no internal tonemapper enabled, to avoid incorrect simulation results. Note that also a tonemapper can be specified, but then optionally disabled (see `tm_enable_tonemapper`) This will steer the firefly filter while not applying the tonemapper to the final image. So a typical workflow would be to tweak the tonemapper settings interactively until the image looks good, and then disabling the tonemapper via `tm_enable_tonemapper` for the final rendering afterwards.

`bool white_mode_enabled = false`
When "white mode" is enabled, Iray Photoreal shades all the elements in the scene (except volumes and the ground fog) using a special diffuse material whose default color is white (its color can be controlled with `white_mode_color`, see below). The user has the option to prevent some material instances from being overridden with the white diffuse material by creating a boolean attribute called `"exclude_from_white_mode"` for those material instances (set to true).

`bool iray_white_mode_enabled = false`
Synonym of the `"white_mode_enabled"` option above.

`mi::Color white_mode_color = mi::Color(0.7)`
Controls the color of the special diffuse material for the white mode.

`mi::Color white_mode_bsdf_weight = mi::Color(0.7)`
Controls the color that will be output in the BSDF weight buffer for the objects with white mode.

`mi::Float32 iray_nominal_luminance = 0`

The nominal luminance is a hint to Iray Photoreal on what is considered a “reasonable” luminance level when viewing the scene. This luminance level is used internally to tune the firefly filter and error estimate. When the nominal luminance value is set to 0, Iray Photoreal will estimate the nominal luminance value from the tonemapper settings. If a user application applies its own tonemapping without using the built-in tonemappers, it is strongly advised to provide a nominal luminance.

Recommendations: For visualization, a reasonable nominal luminance would be the luminance value of a white color that maps to half the maximum brightness of the intended display device. For quantitative architectural daylight simulations (for example, calculating an irradiance buffer or irradiance probes), a reasonable nominal luminance could be the luminance of white paper under average day light.

`mi::IString iray_instancing`
Controls the instancing feature (page 34). Possible values are `"off"`, `"on"`, `"user"`, and `"auto"`. No default is set, which is equivalent to `"off"`.

`mi::IString iray_rt_low_memory`
Controls the memory used by the ray tracing acceleration hierarchies. Possible values are `"on"` and `"auto"`. No default is set, which is equivalent to `"auto"`. For now, this can only reduce memory usage on pre-Turing generation GPUs and the CPU (while potentially harming rendering performance) if `"on"` is used.

`mi::IString shadow_terminator_offset_mode = "user"`
Allows to override the per-object boolean attribute `"shadow_terminator_offset"` globally for all scene objects. Possible values are `"off"`, `"on"`, and `"user"` (the latter will always use the per-object attribute). See “Tessellating curved surfaces” (page 141) for more details about the *shadow terminator artifact* problem. Note that it is not recommended to always have this forced `"on"`, as it can lead to other shadowing problems in some geometric setups.

`mi::IString iray_texture_compression = ""`
Allows to globally override the per-texture texture compression option by setting this attribute to any of the values "off", "medium" or "high". Note that textures used via `tex::lookup_float3` are classified as normal maps internally and only the override value "off" is respected in that case to avoid artifacts (as normal maps do not compress well).

`bool iray_allow_surface_volume_coefficients = true`
Allows the volume scattering and absorption coefficients of materials to be driven by non-uniform MDL expressions other than `base::lookup_volume_coefficients` (which may be used to specify inhomogeneous volumes (page 103)). The expressions will be evaluated once on the surface, i.e. when a ray hits the object, and the volume coefficients will stay the same for subsequent volume interactions within that object. In particular, this enables the usage of 2d surface textures to drive subsurface scattering. Note that in consequence the volumetric properties are typically no longer physically plausible.

4.13 Mixed mode rendering and load balancing

Generally, Iray Photoreal uses all available CPU and GPU resources by default. Iray Photoreal employs a dynamic load balancing scheme that aims at making optimal use of a heterogeneous set of compute units. This includes balancing work to GPUs with different performance characteristics and control over the use of a display GPU.

Iray Photoreal uses one CPU core per GPU to manage rendering on the GPU. When the CPU has been enabled for rendering, all remaining unused CPU cores are used for rendering on the CPU. Note that Iray Photoreal can only manage as many GPUs as there are CPU cores in the system. If the number of enabled GPU resourses exceeds the number of available CPU cores in a system Iray Photoreal will ignore the excess GPUs.

The following key-value pairs, shown here with their default settings, control rendering when passed to `IRender_context::set_option`():

`mi::Float32 ui_responsiveness = 1.f/6.f`

Using the device that drives the screen for rendering with Iray may lead to sluggish UI updates. This can be ameliorated by increasing the value of the `ui_responsiveness` option, up to a maximum of 1. Note, however, that increased UI framerate may come at the cost of severely decreased rendering performance on display devices.

See the `scheduling_niceness` option for details about interactions between the two options.

> **Note:** Currently, this option is supported on Windows only.

`mi::Float32 scheduling_niceness = 0.f`

In situations where more than one render context is running on the same device, agressive scheduling may lead to some contexts only receiving intermittent updates. This is particularly noticeable in situations where a long-running batch render job runs on the same GPU as a render context that is used for navigation.

The same may be true of other processes trying to use graphics or compute resources on a device which is used for Iray Photoreal rendering.

These effects can be ameliorated by increasing the value of the `scheduling_niceness` option, up to a maximum of 1. Note, however, that increasing this option may come at the cost of severely decreased rendering performance, even if no other process is using the same device.

This option has an effect that is very similar to that of the `ui_responsiveness` option, except that it applies to all devices, while `ui_responsiveness` only applies to devices which are currently used to drive displays. Display devices will be scheduled according to the maximum of the two settings, i.e. display devices cannot be scheduled more aggressively than non-display devices.

Note that using `IRendering_configuration::set_resource_enabled`() or the `device_mask` render context option to partition devices between applications or render contexts is generally preferable to increasing `scheduling_niceness`.

`mi::Uint64 device_mask = 0xFFFFFFFFFFFFFFFFull`

The `IRendering_configuration::set_resource_enabled`() function allows control over the set of devices used per *type* of render context, e.g. Photoreal or Interactive. In addition, the `device_mask` render context option allows the set of devices used by a render context *instance* to be restricted beyond what is set via `IRendering_configuration`.

The provided value is interpreted as a bitset. The least significant bit (bit 0) controls use of the CPU for rendering. The remaining bits control GPUs with the same indexing scheme that is used in `IRendering_configuration::set_resource_enabled`().

Note that this option only affects the host on which the render context resides. Since it is much more efficient to partition the machines in a cluster than it would be to use all machines but partition the devices on each, the `cluster_partition` option is provided to control resource usage in network rendering.

`mi::Uint32[ ] cluster_partition = []`
Situations where multiple render contexts share the same cluster, it may be useful to partition the cluster by restricting the set of hosts which each context may use. This may be achieved by setting the `cluster_partition` option. Rendering will be restricted to those active hosts in the cluster which are also listed in the provided array.

An empty array removes all restrictions.

For details about:

- Controlling resource use, see `IRendering_configuration::set_resource_enabled()`.
- Resource scheduling modes, see the section on progressive rendering (page 32).

4.14 Timeout prevention

Iray Photoreal in CUDA mode uses the GPUs in a system to perform rendering. One or more GPUs are often also used to drive the main display device displaying the desktop and applications. On some operating systems CUDA kernel executions are not allowed to exceed a specific amount of time to avoid having an unresponsive user interface.

If a CUDA kernel execution exceeds this limit the operating system may interpret the call as stalled and may restart the driver. To avoid this, Iray Photoreal incorporates a mechanism to automatically adjust the task sizes of GPU function calls so that the total rendering payload per step is divided into smaller fragments depending on the efficiency of the CUDA device on the given scene.

4.15 Progress information

In addition to the areas listed in the general description (page 30), Iray Photoreal may report progress in the following areas if the convergence estimate is active.

`progression`
Reflects the estimated progress between 0 and 1 towards the configured error threshold.

`error`
Reflects the currently estimated average per-pixel error.

4.16 Global performance settings

The following parameters are shared by all render contexts which use Iray Photoreal and are set with the `IRendering_configuration::set_renderer_option()` method.

`iray_nvlink_peer_group_size = "0"`
This setting controls the number of CUDA devices in an NVLINK peer group. Devices in an NVLINK peer group can efficiently share device memory. Iray shares scene memory among all devices in the peer group. This allows Iray to render scenes exceeding the device memory of a single device. Note that devices in the peer group will not share all their scene and rendering related memory (at the moment only bitmap/texture data is shared).

For devices that do not support NVLINK peer access, the option may still enable sharing device memory via PCI Express (if that is supported by the hardware configuration). In this case, however, performance can be significantly lower.

Iray assigns enabled CUDA devices to peer groups in order of provided device id's (see `IRendering_configuration::set_resource_enabled`()). The peer group size needs to be a factor of the total number of enabled CUDA devices. If any two devices assigned to the same group do not have peer access available (e.g. due to the link topology doesn't allow for it), Iray will try to revert to the last provided and valid NVLINK peer group size. Thus, it is the responsibility of the user to make sure that all devices assigned to each group are properly connected through NVLINK.

5 Iray Interactive

Iray Interactive is an interactive raytracing render mode which uses faster yet less accurate rendering algorithms than Iray Photoreal. Iray Interactive targets a look which is consistent with the physically-based result of Iray Photoreal, yet it is optimized for interactive scene manipulations. Iray Interactive leverages NVIDIA CUDA-capable GPUs. It is ideal where ray-tracing effects, such as reflections and refraction, are desired and limited photorealism is acceptable.

The following sections provide an introduction to Iray Interactive, document its system requirements and limitations, and provide further details about capabilities and options.

5.1 Introduction

5.1.1 Purpose

Iray Interactive render mode is aimed at interactive design workflows, with fast feedback for all scene updates, including but not limited to:

- Material and texture editing
- Transforming objects and changing flags
- Placing lights, modifying light type, shape and parameters
- Sun and sky parameters
- Camera parameters (depth of field, tone-mapping, backplate)
- Environment and dome parameters

Note: Vertex-level mesh editing is not yet supported at interactive update rates.

5.1.2 Key Features

The following list highlights key features of Iray Interactive render mode:

Interactivity

- Instant and high-quality visual feedback
- All scene elements editable in realtime

High performance

- Efficient rendering algorithms
- OptiX-based raytracing

Flexibility

- Rendering effects beyond the restrictions of physical plausibility
- Configurable quality/performance dial

Compatibility

- Material Definition Language support
- Plugin renderer mode in NVIDIA Iray
- Cluster rendering

5.2 System requirements

The current system requirements for Iray Interactive:

- The plugin `libirt` needs to be loaded in neuray before this render mode is available. See the `IPlugin_configuration` interface for methods to load plugins.
- Available on Windows x86 64 bit, Linux 64 bit, and macOS 64 bit.
- GPUs of CUDA compute capability 5.0 and higher are supported.
- Driver requirements: The driver needs to support CUDA 11.0.

5.3 Limitations

5.3.1 Result canvases

Iray Interactive supports all render target canvases (page 182) with the following limitations:

- The number of texture spaces available through `texture_coordinate[` *n* `]` is limited to 1.
- Using objects which set the `backplate_mesh` attribute in combination with the dome ground plane will result in undefined behavior, if an additional object is intersecting the dome ground plane.
- Iray Interactive cannot render all light path expressions. Light path expressions including global illumination are not supported, that is, only expressions that capture a subset of `"ES*(D|G)?L"` are supported.
- Iray Interactive cannot render the `irradiance` canvas.
- Iray Interactive cannot render the `motion_vector` canvas.
- Iray Interactive cannot render the `selection_outline` canvas.
- Iray Interactive cannot render the `multi_matte` canvas.
- The effect from normal perturbation functions is not included in the `normal` canvas.
- Canvases held by `IRender_target_opengl` will always be written in their entirety, even if the `ICamera` specifies a window. Rendering will still be limited to the window area but the pixels outside of that area will be set to black.

5.3.2 Materials

The Iray Interactive material model, which is based on MDL ("Material Definition Language" (page 90)), has the following limitations:

- The number of elemental BDSFs per material are restricted to a maximum of 16.
- The number of (procedural) texture functions per material are restricted to a maximum of 32.
- The material volume properties are ignored, so that no volumetric effects are visible except for volume attenuation in glass-like materials.

5.3.3 Rendering

Iray Interactive has the following rendering limitations:

- No volumetric effects except volume attenuation by Beer's law.
- Emissive geometry is not illuminating the scene, but it is considered when the geometry is hit directly.
- Non-specular refraction of finite lights is rendered black.
- No motion-blur.
- No camera aperture function support.
- No custom matte objects.
- No ghostlights.
- If the `irt_ambient_shadow_mode` is not equal to 1, that is, when environment shadows are off or when screenspace approximation is used, diffuse roughness is not supported and subtle lighting differences can be visible if the dome is in spherical or hemispherical mode. Some stronger lighting differences may be visible when using measured materials.
- Environment lighting of diffuse layers does not take into account the dome mode, the dome texture scale parameter and the position and size of the dome. The light is always collected from the infinite environment, applying only the dome rotation parameters.
- The scene option `environment_dome_ground_connect_from_camera` is not supported.
- The shadow buffer does not contain glossy reflections of shadow computations on the virtual ground plane.
- Triplanar projectors are not correctly displayed in the `texture_coordinate` canvas.
- Automatic normal clamping (to prevent darkened materials due to strong perturbation of shading normals) is slightly different from Iray Photoreal.
- Iray Interactive ignores the following attributes:
  - `reflection_cast`
  - `reflection_recv`
  - `refraction_cast`
  - `refraction_recv`
  - `face_front`
  - `face_back`
- The attribute `backplate_mesh_function` is not supported.

- The camera's fisheye lens distortion mode (`mip_lens_distortion_type = "equidistant"`) is not supported.
- Section cap color can be set globally via the scene options, but not individually via `ISection_object::set_cap_color`.

5.3.4 Picking

Picking reports only the first hit as a result.

Picking rectangles (i.e., more than one ray) is not supported.

5.3.5 Multi-GPU Configurations

NVLINK is not supported in Iray Interactive.

Due to prohibitive inter-GPU synchronization traffic, the following options are not available in multi-GPU configurations:

- "Path space filtering" (page 74)
- FXAA antialiasing, see the rendering options of Iray Interactive

5.3.6 Instancing

The "user" instancing mode is not supported in Iray Interactive. The "auto" instancing mode is supported.

5.3.7 Fibers

Fiber geometry is not supported in Iray Interactive.

5.3.8 Mixing GPU and CPU rendering

Unlike Iray Photoreal, Iray Interactive is not able to do hybrid/mixed rendering on both GPUs and the CPU.

5.3.9 CPU fallback

Unlike Iray Photoreal, Iray Interactive is not able to finalize the rendering of canvases that have been started on GPUs via its CPU fallback (due to missing hybrid/mixed rendering). So in case of a GPU failure, the current rendering has to be restarted to render on the CPU.

5.4 Progressive rendering

Iray Interactive supports progressive rendering into a render context.

On each render call the image will be refined more so that over time the image converges. The intermediate images can be displayed to the user thus giving a good early impression about the lighting and material look of the scene. The renderer makes some approximations aiming to produce an image with no or very little noise on exterior, turntable-type scenes.

For each render request, the renderer will automatically check if the parameters for the requested image (for example image resolution or camera position) match to the last rendered image. Also the scene will be checked for changes. If any changes are found that would result

in visible changes in the image the render context will be reset and the progressive rendering starts from the beginning.

The following attributes on the `IOptions` class, shown here with their default settings, control progressive rendering:

`mi::Sint32 progressive_rendering_max_samples = 100`
: The maximum number of samples is compared to the number of samples since the beginning of this progression. The render call will return 1 if the rendering loop is terminated by this termination criterion.

  A value of -1 will disable the criterion altogether.

`mi::Sint32 progressive_rendering_max_time = 3600`
: The maximum time in seconds is compared to the rendering time since the beginning of this progression. The render call will return 1 if the rendering loop is terminated by this termination criterion.

  A value of -1 will disable the criterion altogether.

`bool progressive_rendering_quality_enabled = true`
: Controls rendering termination based on a quality estimate. Note that in the Interactive mode, this estimate is only available together with the `progressive_rendering_quality_ssim` option.

`bool progressive_rendering_quality_ssim = false`
: Enables a convergence quality estimate based on the SSIM predictor (page 39). Rendering will stop once the configured `post_ssim_predict_target` is reached. Note that the predictor must be enabled for this option to work.

5.4.1 Multi-GPU scalability and automatic sample count adaption

Similarly to Iray Photoreal, Iray Interactive supports both interactive and batch modes. In the interactive mode the renderer aims at providing the highest frame rate. The batch mode focuses on obtaining a converged image as fast as possible by rendering many iterations simultaneously, thus making best use of the GPU resources. The gain is particularly visible on multi-GPU machines and network rendering.

The scheduling mode can be passed to `IRender_context::set_option`:

`const char* scheduler_mode = "interactive"`
: Controls the rendering mode, which is by default interactive. Setting this option to “batch” switches to the non-interactive rendering mode.

In interactive mode, the tradeoff between interactivity and convergence speed can be controlled manually. Please note that the following options have no effect when batch scheduling is enabled.

Iray Interactive enables an automatic sample count adaption. This is controlled with the `irt_automatic_sample_count` scene option.

`bool irt_automatic_sample_count = false`
: If enabled, using multiple GPUs will not lead to a higher interactivity, but to faster convergence. Since high parallelism is sustained by adapting the amount of work

computed in each frame to the available resources, the resulting scalability is improved drastically.

In some cases, this behavior can be too extreme. The trade-off between scalability and interactivity can also be controlled manually by adapting the number of samples per frame with the `progressive_rendering_samples` rendering option.

`mi::Sint32 progressive_rendering_samples = 1`

This sample count is taken into account if the automatic sample count is disabled. For instance, when rendering in the network on 2 hosts, each equipped with 4 GPUs, a reasonable choice would be to set the number of samples to 4, leading to higher interactivity (max. 2x) and faster convergence (max. 4x).

As screen space shadow approximations and image filtering can degrade multi-GPU scalability drastically, those are only available in single-GPU mode.

5.5 Lighting

Lighting features of Iray Interactive render mode include:

- Ray-traced, accurate hard- and soft-shadows
- Accurate reflections and refractions
- Efficient direct light sampling, including light profiles
- Efficient multi-scale environment sampling. Initial coarse-scale approximation with low noise level converges quickly over time to an accurate result.
- Approximated indirect lighting (diffuse and glossy but diffuse-only on primary hit).
- Fast, smooth approximation of matte ground shadows
- Two ambient-occlusion modes (diffuse AO, AO guided by environment)

5.6 Rendering options

The following sections describe specific attributes on the `IOptions` class, shown here with their default settings, for ambient occlusion and shadows as well as other miscellaneous parameters.

5.6.1 Ambient occlusion and lighting

When indirect lighting is not enabled, Iray Interactive determines the incoming lighting at a point as the sum of the contributions of the light sources, of the environment, and of an approximation of the indirect lighting. The contribution of the environment can be either obtained by the environment itself, or approximated by a constant color. This constant color can also be modulated by ambient occlusion. The indirect lighting is approximated by a constant color, also potentially modulated by ambient occlusion.

`mi::Sint32 irt_env_lighting_mode = 0`

- `0` — Environment lighting
- `1` — Constant color approximation
- `2` — Ambient occlusion approximation

This setting determines how the computation of environment lighting is performed. Using the default value of 0, Iray Interactive computes the environment lighting from the actual radiance coming from the environment. The value 1 approximates the contributions of the environment as `irt_env_approx_color`. The last value triggers the computation of ambient occlusion. The behavior of Iray Interactive is tied to the `irt_ambient_occlusion_mode` value. When `irt_ambient_occlusion_mode` is 0, the environment is considered uniform, with an emission determined by `irt_env_approx_color`. Otherwise, the emission of the environment is the monochromatic luminance of the actual environment.

Note that this setting only affects the lighting on diffuse and glossy surfaces, while the actual environment is always used when computing specular reflections.

`mi::Color irt_env_approx_color = 0`

Intensity of the constant lighting used to approximate environment lighting when `irt_env_lighting_mode` is not 0.

`mi::Sint32 irt_ambient_occlusion_mode = 0`

- `0` — Diffuse ambient occlusion
- `1` — Environment-guided ambient occlusion

This setting only affects the `ambient_occlusion` output canvas and the beauty render when `irt_use_ambient_occlusion` is `true`. The first variant corresponds to the common diffuse ambient occlusion calculation, which integrates the visibility weighted by the cosine of the direction and the normal over the hemisphere. The difference between the two settings is that the first variant assumes uniform ambient lighting, whereas the second lights the scene under the actual environment used in the scene. The exact behavior differs depending on the rendering output:

`ambient_occlusion` output

Both methods render the scene with a white Lambertian material. The environment-guided ambient occlusion results in a color output.

`beauty` output

The scene is rendered with full material evaluation, but the contribution of the environment is computed from a user-controlled intensity when `irt_ambient_occlusion_mode` is 0 and from monochromatic environment luminance otherwise.

`mi::Float32 irt_environment_occlusion_scale = 1.0`

This setting describes the overall ratio between the openings and occlusions in the scene. The environment lighting is then dimmed accordingly instead of relying on actual environment lighting. Fast convergence with consistent lighting can then be achieved using IBL falloffs.

`bool irt_use_ambient_occlusion = false`

This setting enables the use of ambient occlusion as an approximation of indirect lighting, where the approximated indirect lighting is defined by `irt_ambient_intensity`.

`mi::Color irt_ambient_intensity = 0`

Intensity of additional ambient lighting, typically used to approximate global illumination effects.

`mi::Float32 ambient_falloff_max_distance = FLT_MAX`
World space distance beyond which potential occlusion does not influence result of ambient lighting and AO, as well as IBL if `irt_ibl_falloff` is `true`.

`mi::Float32 ambient_falloff_min_distance = FLT_MAX`
World space distance below which all potential occlusion fully influences result of ambient lighting and AO, as well as IBL if `irt_ibl_falloff` is `true`.

`mi::Float32 ambient_falloff = 1`
Exponent to be used for weighting the occlusion value between falloff min and max distance. The default 1 blends linearly.

`mi::Sint32 irt_ambient_falloff_distance_space = 0`

- `0` — World space
- `1` — Raster space (pixels)

Defines whether the `irt_ambient_falloff_min,max_distance` values correspond to world space or raster space.

`mi::Sint32 irt_fast_convergence_start = -1`
Defines the iteration number after which the iteration speed is reduced in favor of more efficient convergence, saving up to 30 percent of the total render time to a converged image. The default value of -1 disables this optimization altogether.

Note that when fast convergence is enabled, for a given iteration number the images will slightly differ from the image obtained with fast convergence disabled.

`mi::Sint32 irt_fast_convergence_ramp_up = 0`
Defines the number of iterations over which speed is gradually reduced in favor of more efficient convergence. The default value of 0 means that the speed reduction happens right away at the iteration specified with the `irt_fast_convergence_start` option.

5.6.2 Indirect lighting

Iray Interactive includes a fast approximation technique for indirect lighting. This technique takes into account all light sources, HDRI environment, physical sun and sky and diffuse and glossy components of the material.

Iray Interactive supports two indirect illumination modes. The coarse mode is suitable to obtain a fast early impression of the indirect illumination in the scene. At the first primary hit point, the indirect illumination does only contribute to the diffuse layers, which means that glossy contributions are only taken into account at secondary hits and beyond.

In fine scale mode, all reflective layers, including glossy and measured BRDFs, of the material receive indirect illumination. Furthermore, indirect illumination from the close vicinity of the shading location is taken into account more accurately. In contrast to the coarse scale approximation, this mode does not introduce any undesirable corner darkening.

In turntable like outdoor scenes, indirect lighting can often be neglected, but it is usually an important contribution in indoor scenes.

Note: The lighting and shadowing approximation values described in this section are automatically disabled with a warning message when `irt_indirect_light_mode` is not 0.

`mi::Sint32 irt_indirect_light_mode = 0`

- `0` — No indirect lighting except specular transmission and reflection
- `1` — Coarse scale diffuse indirect illumination
- `2` — Fine scale indirect illumination, including glossy and measured BRDFs

`bool irt_indirect_outlier_rejection = true`

The technique automatically rejects outliers which would lead to displeasing low-frequency contributions which converge away only slowly. This is similar to the firefly filter in Iray Photoreal. However, this means that some contributions are cut-off, which can be disabled by setting the above option to `false`.

Fig. 5.1 - Simple indoor scene without (0), with coarse scale (1) and with fine scale (2) indirect lighting enabled.

5.6.3 Shadows

`bool irt_area_as_point_lights = false`

Approximate area by point lights, for example to avoid shadow noise during interaction.

Fig. 5.2 - Simple scene rendered with area lights approximated by point lights (left) and full area light sampling (right)

`mi::Sint32 irt_ambient_shadow_mode = 1`

Allows to control occlusion tests for environment, instant sun and sky and ambient lighting:

- `0` – No occlusion
- `1` – Raytraced occlusion
- `2` – Screen space approximation, with directional occlusion tests for non-diffuse materials. This mode is less precise than raytraced occlusion tests, but converges faster to a noise free result. Noise results mainly from glossy components of the material. View-dependent occlusion seen in glossy layers, such as seen on the ground floor of the image, are retained.
- `3` – Screen-space ambient occlusion applied to all material layers. In this mode, less noise is visible on glossy materials to the expense of losing directional effects in the occlusion.

In modes 2 and 3, the influence region is controlled by the `irt_ambient_falloff_min,max_distance` parameters and the radius is always clamped to not exceed 200 pixels in raster space. In raytraced mode, this is only the case when `irt_ibl_falloff` is `true`, otherwise the occlusion distance is unlimited.

Since ambient and IBL shadows can be turned off by setting the value to 0, the parameter `irt_ibl_shadows` is now deprecated.

Note that modes 2 and 3 cannot be used in a multi-GPU setup for performance reasons.

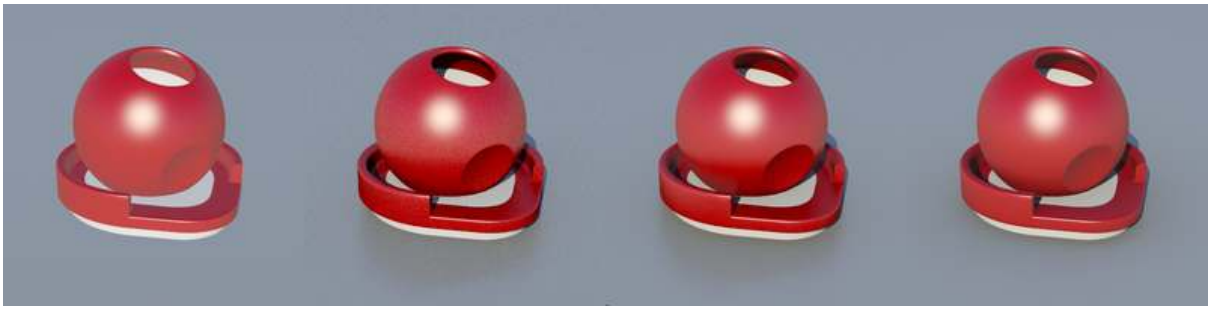


Fig. 5.3 - Simple scene rendered with modes 0, 1, 2 and 3

`bool irt_shadows = true`
Allows shadows to be forcibly turned off, independent of scene shadow flags.

`bool irt_ibl_falloff = false`
If this flag is enabled, the irt_ambient_falloff_min/max_distance parameters also affect raytraced occlusion for IBL and instant sun and sky.

`bool irt_ground_shadow_filter = true`
Allows the filtering of the ground shadows of the implicit ground plane to be turned off.

`bool shadow_cast = true`
The per-instance bool shadow_cast flag controls whether an object is casting a shadow. It is taken into account by the renderer if `irt_shadows` is enabled.

`bool_shadow_recv = true`
The per-instance bool shadow_recv flag controls whether an object will display the effect of shadows cast by other objects (“receives” the shadow). It is taken into account by the renderer if `irt_shadows` is enabled.

5.6.4 Additional approximations

Iray Interactive allows to trade quality against performance in various ways.

`mi::Sint32 depth = 16, mi::Sint32 depth_reflect = 4, mi::Sint32 depth_refract = 4, mi::Sint32 depth_shadow = 16, mi::Sint32 depth_cutout = 16`

Maximum trace depth settings are respected by Iray Interactive. These values correspond to the maximum total number of bounces, maximum number of reflections and refractions, maximum number of shadow refractions and maximum depth for cutouts respectively. The maximum total number of bounces does not influence the cutout depth.

Fig. 5.4 - From left to right: A glass object rendered with different trace depths 8, 6, 4 and 2

`mi::Float32 irt_ray_importance_threshold = 0.02`

If the contribution of a specular reflection or refraction bounce falls below this value it is cut off. The lower the value the more accurate the result. The tradeoff is elapsed time. Compared to limiting the trace depth, using this setting usually results in a better performance-quality ratio.

Fig. 5.5 - From left to right: A glass object rendered with importance thresholds 0, 0.05, 0.1 and 0.4

`mi::Sint32 irt_render_mode = 1`

When set to 0, the renderer is in interactive mode, depth of field and IBL shadows are turned off and some additional approximations are used. When set to 1, the standard refinement mode is used.

5.6.5 Antialiasing and filtering

Iray Interactive supports the filter parameter in the same way as Iray Photoreal (see Rendering options (page 56) in Iray Photoreal).

`bool irt_first_frame_antialiasing = false`

FXAA antialiasing is applied by default to the first frame when `irt_render_mode` is set to zero, improving in particular the appearance of sharp edges. FXAA is not available in multi-GPU configurations.

`mi::Sint32 filter = FILTER_BOX`

Iray Interactive natively supports `FILTER_BOX`, `FILTER_TRIANGLE` and `FILTER_GAUSS` filters for antialiasing. If `FILTER_CMITCHELL` or `FILTER_CLANCZOS` is requested, Iray Interactive will use the default box filter instead.

`mi::Float32 radius`
The radius of the filter kernel. Recommended values are 0.5 for box filter, 1 for triangle filter and 1.5 for Gauss filter.

`bool iray_firefly_filter = true`
Controls a built-in filter to reduce bright spots that may occur under some difficult lighting conditions. Such bright undesired pixels are often called "fireflies."

The filter works best in combination with the built-in tonemappers. If a tonemapper is not enabled, the filter estimates can be too pessimistic for some scenes, resulting in some fireflies to still appear.

`mi::Float32 iray_nominal_luminance = 0`
The nominal luminance is a hint to Iray Interactive on what is considered a "reasonable" luminance level when viewing the scene. This luminance level is used internally to tune the firefly filter and error estimate. When the nominal luminance value is set to 0, Iray Interactive will estimate the nominal luminance value from the tonemapper settings. If a user application applies its own tonemapping without using the built-in tonemappers, it is strongly advised to provide a nominal luminance.

> **Recommendations:** For visualization, a reasonable nominal luminance would be the luminance value of a white color that maps to half the maximum brightness of the intended display device. For quantitative architectural daylight simulations (using the irradiance buffer), a reasonable nominal luminance could be the luminance of white paper under average day light.

5.6.6 Environment map resolution

`mi::Sint32 irt_environment_max_resolution = 2048`
This parameter sets the maximum HDRI environment map resolution (width) used by the renderer. All environment maps with a higher resolution are scaled down to this size. This avoids high preprocessing times and device memory consumption for large environment maps to the expense of higher image quality.

5.6.7 White mode

When "white mode" is enabled, Iray Interactive shades all the objects in the scene using a special diffuse material whose default color is white (its color can be controlled with `white_mode_color`, see below). The user has the option to prevent some material instances from being overridden with the white diffuse material by creating a boolean attribute called `"exclude_from_white_mode"` for those material instances (set to true).

`bool white_mode_enabled = false`
Enables the white mode.

`mi::Color white_mode_color = mi::Color(0.7)`
Controls the color of the special diffuse material for the white mode.

`mi::Color white_mode_bsdf_weight = mi::Color(0.7)`
Controls the color that will be output in the BSDF weight buffer for the objects with white mode.

5.7 Path space filtering

Path space filtering (PSF) reduces image noise in Iray Interactive by selective averaging of the shading results obtained in neighboring pixels. Over time the amount of averaging is reduced, so that no permanent artifacts will be introduced to the final image.

The following attributes on the `IOptions` class, shown here with their default settings, control path space filtering:

`bool irt_psf_enable = false`
Enables path space filtering if set to true.

`mi::Float32 irt_psf_geometry_filter = 1.0`
This parameter controls the influence of geometric distance in the path space filter. The recommended range is from 0.0 to 1.0. At 0.0, path space filtering is disabled. The default 1.0 represents the recommended best choice for this geometric distance influence. If geometric details are too much blurred or disappear in initial frames, this parameter can be reduced, which will preserve more geometric detail at the cost of additional noise.

`mi::Float32 irt_psf_lighting_filter = 1.0`
This parameter controls the influence of lighting differences in the path space filter. The recommended range is from 0.0 to 1.0. At 0.0, path space filtering is disabled. The default 1.0 represents the recommended best choice for this lighting difference influence on the filter. If shadow boundaries are too much blurred or disappear in initial frames, this parameter can be reduced, which will preserve more shadow boundary detail at the cost of additional noise.

`mi::Sint32 irt_psf_convergence_frame = 50`
To ensure convergence the similarity thresholds are automatically reduced over time to avoid persistent filtering artifacts. The value of this parameter indicates the frame index at which the thresholds reach their minimal value. Further frames are then minimally filtered.

Path space filtering stores temporary light path information and increases the memory demands on the GPU proportional to the image size. Both, memory and performance overhead are independent of the scene size and shading complexity making PSF even more beneficial for complex renderings.

Note: Using PSF on multiple GPUs would generate high inter-GPU synchronization traffic, resulting in a significant loss of performance. PSF is thus automatically disabled on multi-GPU configurations.

5.8 Global performance settings

The following parameters are shared by all render contexts which use Iray Interactive and are set with the `IRendering_configuration::set_renderer_option` method.

`mi::Float32 irt_working_memory = "0.9"`
This setting limits the amount of working device memory available to Iray Interactive as a fraction of the total device memory. The working device memory is used internally for temporary storage. It does not include memory required for scene and frame buffer storage. The setting is only used as an upper bound. Iray will allocate as much free device memory as possible up to the amount specified. By default the amount is set to 90increased working memory so his option should be set as high as possible.

5.9 Comparison with Iray Photoreal

Following is a list of key differences with Iray Photoreal:

- Physically based but not unbiased.
- Currently, no support for:
  - Volumetric effects
  - Non-specular refraction (frosted glass appears black)
  - Motion blur
  - Custom matte objects
  - Full light path expressions
- Support for physically implausible flags:
  - *Objects* — `shadow_cast`, `shadow_recv` Shadows are missing if not enabled in the scene.
  - *Lights* — `shadow_cast`
- Support for trace depth parameters: `depth`, `depth_reflect`, `depth_refract`, `depth_shadow`
- Differences in behavior of directional lights and shadows when the environment dome is enabled.
- The opacity of decals can only be controlled by the cutout value, but not semi-transparent BSDFs. Furthermore, decals are not taken into account for shadows.

6 Scene database

The following sections introduce the scene database element interfaces and an example program that creates a scene programmatically by making calls to the Iray API.

6.1 Overview of the scene database elements

The following is a list of the `Scene elements` and the primary classes that implement them, with links to detailed information about these interfaces.

6.1.1 Scene graph structure

`IInstance`
A scene element that adds a transformation and attributes to another scene element. Every instance references exactly one scene element.

`IGroup`
A container for other scene elements. Typically used to structure scene elements.

See also the section `Scene graph structure` in the API reference.

6.1.2 Leaf nodes

`ICamera`
The viewpoint from which the scene is rendered. The camera points down the negative z-axis.

`ILight`
A light source such as a spot, a directional, or an area light. A light points down the negative z-axis.

`IOptions`
Scene-specific settings stored as a scene element. Settings are typically exposed as attributes.

`ITriangle_mesh`
A triangle mesh, which is a type of polygon mesh. Incremental construction of triangle meshes is supported.

`IPolygon_mesh`
A polygon mesh, which defines the shape of a polyhedral object. Incremental construction of polygon meshes is supported.

`ISubdivision_surface`
A subdivision surface, which is a smooth surface represented by a coarse control mesh.

`IFreeform_surface`
A free-form surface, which is used to represent objects from smooth patches with (optional) trimming curves. Typically, a free-form surface object consists of several surfaces.

`IOn_demand_mesh`
An on-demand mesh, which describes a triangle mesh in a much simpler format and is provided on a user-specified callback. Often, this data is only needed temporarily.

See also the section `Leaf nodes` in the API reference.

6.1.3 MDL-related elements

`IFunction_call`
A function call, which is an instance of a formal function definition with a fixed set of arguments.

`IMaterial_instance`
A material instance, which is a concrete instance of a formal material definition with a fixed set of arguments.

See also the section `MDL-related elements` in the API reference.

6.1.4 Miscellaneous classes

`IAttribute_container`
A container that stores attributes only.

`IBsdf_measurement`
A scene element that stores measured BSDF data. A BSDF measurement typically appears in a scene as an argument of an MDL material instance.

`IDecal`
A decal, which is a sticker-like object that can be applied to other geometry objects.

`IImage`
A pixel image file, which supports different pixel types.

`IIrradiance_probes`
An irradiance probe, which is used to render the irradiance at certain locations in a scene.

`ILightprofile`
A light profile, which is a digital profile of a real world light. It typically appears in a scene as an argument of an MDL material instance.

`ITexture`
A texture, which adds image processing options to images. It typically appears in a scene as an argument of an MDL function call or MDL material instance.

6.2 Creating a scene programmatically

This section introduces:

1. Core concepts about creating a scene by using API calls. Alternatively, a scene can be created by importing a scene file.
   - Comparison to scene file main.mi (page 78)
   - Order of scene element creation (page 78)
   - Procedure for scene element creation (page 78)
   - Categorizing scene elements (page 79)
2. Example source file `example_scene.cpp` (page 297), which demonstrates how to create a scene by using calls to the Iray API

6.2.1 Comparison to scene file main.mi

The example program `example_scene.cpp` (page 297) is very similar to `example_rendering.cpp` (page 282). The key difference is that `example_scene.cpp` (page 297) does not import a scene file; it calls the function create_scene() which constructs a scene programatically by using calls to the Iray API.

This programmatically created scene is almost identical to the scene which is created by importing `main.mi`. The key difference is that some features which are not needed in `example_rendering.cpp` (page 282) have been ommitted in `example_scene.cpp` (page 297) (some MDL material instances and the texture coordinates for the ground plane). The only file loaded from disk is `main.mdl`, which contains the MDL definitions for the materials used in `example_scene.cpp` (page 297).

6.2.2 Order of scene element creation

The various scene elements are created one-by-one in a bottom-up order (this is the same order as in the `main.mi` file). Order is important as soon as references between database elements are created.

A referenced DB element must already exist in the database when the reference is set. If a top-down order is needed for some reason, you must delay setting the reference until after the referenced database element is stored in the database.

6.2.3 Procedure for scene element creation

Creating a certain scene element is typically a three step procedure:

1. Create a scene element by calling `ITransaction::create`.
2. Set up the properties of the scene element, including any attributes.
3. Use `ITransaction::store` to store the scene element in the database under a name that can be used later to re-access or to reference it.

> **Note:** MDL material instances and functions deviate from this scheme because they require the use of a dedicated factory function for the corresponding material or function definition.

6.2.4 Editing scene topology

After the scene elements have been created, the internal structure of the scene — its *topology* — can also be modified. However, the effect of defining a new camera with `IScene::set_camera_instance` on render contexts that already exist is undefined and may vary between render modes. Instead, the current camera instance, acquired with `IScene::get_camera_instance`, should be edited. Alternatively, all current render contexts of the scene can be destroyed and recreated with the new camera instance. In all scenes, however, the there must be a path from root group to the camera instance.

6.2.5 Categorizing scene elements

The different scene elements can be categorized as follows:

- Structural elements. Examples: instances and groups
- Leaf nodes. Examples: cameras, lights, geometry, and options
- MDL-related elements. Examples: materials and functions
- Miscellaneous elements. Examples: images, textures, light profiles, and BSDF measurements

The details of the various scene element types is beyond the scope of this example. For additional details, see the examples for MDL in `example_mdl.cpp` (page 230) and triangle meshes in `example_triangle_mesh.cpp` (page 316).

6.2.6 Example – Creating a scene through the API

Source code
`example_scene.cpp` (page 297)

The example creates a scene through calls to the API, renders the scene and writes the rendered image to a file.

7 Geometry

The following sections describe geometric concepts and examples of their use.

> **Note:** Iray uses a right-handed Cartesian coordinate system internally.

7.1 Creating triangle meshes

The following sections introduce core concepts about creating triangle meshes, storing them as database elements, adding them to a scene graph, and retrieving and editing them:

- Creating a triangle mesh (page 80)
- Adding geometry to a scene (page 80)
- Retrieving and editing triangle meshes (page 81)

The program `example_triangle_mesh.cpp` (page 316) serves as an example implementation of these concepts.

7.1.1 Creating a triangle mesh

To create a triangle mesh, you need to specify at least the following:

- The points (the position of the vertices) of the triangle mesh
- The triangles (as point indices)

In `example_triangle_mesh.cpp` (page 316), function create_tetrahedron is used to create a tetrahedron with four points and four triangles.

Vertex normals are an attribute of the triangle mesh. In contrast to generic methods for attributes supported by `IAttribute_set`, meshes provide their own methods to enable access to mesh-specific attributes. In `example_triangle_mesh.cpp` (page 316), one normal per point is specified; hence, the mesh connectivity is used to create and attach the attribute vector.

7.1.2 Adding geometry to a scene

After geometry has been created and stored as a database element, it is necessary to include it in the scene graph (unless you do not want it to be part of the scene). The most common approach is to create an instance node that instantiates the geometry, and to include that instance in some group, for example the root group. The instance node allows you to share the geometry between several instances while having different settings per instance. For example, different instances typically have different transformation matrices, and might have different attributes, for example different materials.

In `example_triangle_mesh.cpp` (page 316), function `^setup_scene` is used to create an instance of each mesh and to set the transformation matrix and the `visible` and `material` attribute. Both instances are then added to the root group.

7.1.3 Retrieving and editing triangle meshes

All triangle mesh data can be retrieved and changed by using the API. The example program `example_triangle_mesh.cpp` (page 316) uses a Loop-subdivision scheme for triangle meshes to subdivide a copy of the tetrahedron created previously.

It is possible to retrieve and change the number of points and triangles, as well as the point coordinates or triangle indices. To access the mesh-specific attributes, you must acquire the corresponding attribute vector. If you have obtained a non-const attribute vector you must re-attach it to the mesh after you are finished with it.

7.1.4 Example - Using triangle meshes

Source code
`example_triangle_mesh.cpp` (page 316)

This example creates and manipulates triangle meshes. The MDL search path is specified by a command-line argument. The rendered image is written to file.

7.2 Creating polygon meshes

The following sections introduce core concepts about creating and tessellating polygon meshes:

- Creating a polygon mesh (page 81)
- Tessellating a polygon mesh (page 82)
- Example - Adding a polygon mesh to a scene (page 82)

The program `example_polygon_mesh.cpp` (page 259) serves as an example implementation of these concepts. This program imports a partial scene containing definitions for the camera, a light, and some geometry (a ground plane and a yellow cube). Using calls to the API, it creates a red cylinder as a polygon mesh and tessellates a copy of it.

7.2.1 Creating a polygon mesh

To create a polygon mesh, you need to specify at least the following:

- Specify the points (the position of the vertices) of the polygon mesh
- Polygons (as point indices)

In `example_polygon_mesh.cpp` (page 259), function `create_cylinder` is used to create a cylinder with a regular N-gon as base. The actual data such as point coordinates and normal directions is not hard-coded, but computed by the helper class `Cylinder` and based on a few input parameters: radius, height, and parameter N of the N-gon.

To get a sharp edge for the top and base face, vertex normals are not specified per point, but per vertex. Hence, the mesh connectivity cannot be used. A custom connectivity must be used instead.

> **Note:** In the previous example, "Creating triangle meshes" (page 80), vertex normals are specified per point and the mesh connectivity is used to define vertex normals.

The vertices are mapped to entries in the attribute vector as follows:

- All vertices of the top face are mapped to the same normal
- All vertices of the base face are mapped to the same normal
- Each two vertices on the edge between two adjacent side faces share one normal

7.2.2 Tessellating a polygon mesh

The `ITessellator` class is a *functor* that tessellates a polygon mesh and returns the tessellated mesh as a triangle mesh. In `example_polygon_mesh.cpp` (page 259), the red cylinder is tessellated and the result is added to the scene (with a blue material). The tessellator does not support any options to control its behavior.

7.2.3 Example - Adding a polygon mesh to a scene

Source code
`example_polygon_mesh.cpp` (page 259)

This example demonstrates how to add a polygon mesh to a scene graph. For more information, see "Adding geometry to a scene" (page 80).

7.3 Creating on-demand meshes

On-demand meshes are created using the `ISimple_mesh` interface in two steps:

1. Set the callback object that returns an instance of `ISimple_mesh`. This instance holds the actual geometry data.
2. Set the bounding box and the maximum displacement.

7.3.1 Example - Using on-demand meshes

Source code
`example_on_demand_mesh.cpp` (page 248)

This example demonstrates an implementation of the abstract `ISimple_mesh` interface. The program imports a partial scene containing definitions for the camera, a light, and some geometry (a ground plane and a yellow cube). It then replaces the triangle mesh of the yellow cube by an equivalent on-demand mesh.

This particular implementation represents fixed geometry: a unit cube centered at the origin with face normals. In a typical application, this would probably be an adapter that adapts the application-specific geometry format to the format expected by `ISimple_mesh`.

> **Note:** The `Cube` class is such an adapter: it adapts the original data stored in `m_points` and `m_normals` with their own index arrays to the data arrays expected by the `ISimple_mesh` interface with a single index array `m_triangles`.

7.4 Creating subdivision surfaces

The following sections introduce core concepts about the control mesh of a subdivision surface and the instancing of objects:

- The control mesh (page 83)

- Instancing objects (page 83)

The program `example_subdivision_surface.cpp` (page 311), which serves as an example implementation. The program imports a partial scene containing definitions for the camera, a light, and some geometry (a ground plane and a yellow cube). Using the API, it then creates a subdivision surface and three instances of it with different approximation levels.

7.4.1 The control mesh

The control mesh of the subdivision surface is exposed as a polygon mesh with the limitation that only triangles and quads are supported. (See `example_polygon_mesh.cpp` (page 259) for an example for polygon meshes.) In `example_subdivision_surface.cpp` (page 311), a simple cube is created. Additionally, the crease values of the edges of two opposing faces are set to 1.0, which creates sharp edges in the limit surface (which is a cylinder). Without the crease values the limit surface would be a sphere.

7.4.2 Instancing objects

The procedure to add the subdivision element to the scene database and the scene graph is similar to the procedure previously used for the tetrahedron in `example_triangle_mesh.cpp` (page 316) and the cylinder in `example_polygon_mesh.cpp` (page 259).

In the earlier examples, one instance of the element was created. In `example_subdivision_surface.cpp` (page 311), three instances of the subdivision element are created. Multiple identical instances share the same geometry. Each instance has its own transformation matrix and each instance can have its own attributes. To show the subdivision process in `example_subdivision_surface.cpp` (page 311), different approximation levels are used for each instance.

7.4.3 Example - Using subdivision surfaces

Source code

`example_subdivision_surface.cpp` (page 311)

This example creates three instances of a subdivision surface with different approximation levels. The MDL search path is specified by a command-line argument. The rendered image is written to file.

7.5 Creating free-form surfaces

The following sections introduce core concepts related to the `IFreeform_surface` interface and the use of the transform matrix of a sphere to create an ellipsoidal shape:

- Creating a free-form surface (page 84)
- Adding free-form objects to a scene (page 84)

The program `example_freeform_surface.cpp` (page 217) serves as an example implementation. The program imports a partial scene containing definitions for the camera, a light, and some geometry (a ground plane and a yellow cube). Using the API, it then creates two free-form surfaces and adds them to the scene.

7.5.1 Creating a free-form surface

In `example_freeform_surface.cpp` (page 217), the function create_sphere is used to create a free-form surface that represents a sphere. A representation as Bezier surface of degree 2 in both directions is used. The sphere is centered at the origin; its radius is 1.0.

7.5.2 Adding free-form objects to a scene

In `example_freeform_surface.cpp` (page 217), two free-form objects are added to the scene: a red sphere, and a blue partial ellipsoid. The free-form surface of the blue ellipsoid represents a sphere; the ellipsoidal shape is created by using its transformation matrix. The parameter range in U-direction is restricted to [0.0, 0.5] to cut away the part of the ellipsoid that lies below the ground plane.

7.5.3 Example - Using free-form surfaces

Source code

`example_freeform_surface.cpp` (page 217)

This example creates three instances of a free-form surface with different approximation levels. The MDL search path is specified by a command-line argument. The rendered image is written to file.

7.6 Creating fibers

The following sections introduce core concepts related to the `IFibers` interface:

- The geometry of a fiber (page 84)
- The IFibers object (page 86)
- Adding a single fiber to the IFibers object (page 87)
- Example - Creating fibers (page 87)

7.6.1 The geometry of a fiber

Fibers are represented by three-dimensional curves, specified by a set of control points in object space. The radius of a cylindrical cross-section defines the thickness of the fiber. This radius can be specified for the length of the entire fiber or for each control point.

Three types of curves are supported: linear curves, uniform cubic B-splines and Catmull-Rom curves.

In case of uniform cubic B-splines and Catmull-Rom curves a minimum of four control points are required per fiber and define the first segment. Each additional control point defines a new segment, as three control points are shared with the previous segment. Note that repetition of identical vertices (also known as pinned curves), for example to enforce a B-spline to pass through a vertex, is not supported and should rather be achieved using 'phantom vertices'. Catmull-Rom curves start at the second control point and intersect all subsequent control points except for the last one.

In case of linear curves a minimum of two control points are required per fiber and define the first segment. Each additional control point defines a new segment, as one control point is shared with the previous segment.

The methods of the interface specify whether attributes are assigned to the primitive (the fiber instance) or to each control point. This assignment is defined using the following `enum` values:

`FIBER_ATTRIBUTE_PER_PRIMITIVE`

`FIBER_ATTRIBUTE_PER_VERTEX`

7.6.2 The fiber control points

The control points of a fiber define the shape of the curve. Figure 7.1 is a B-spline curve with fourteen control points:

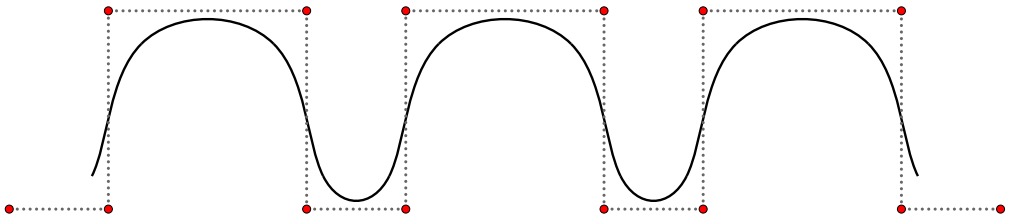


Fig. 7.1 - The control points (in red) shape the B-spline curve

Note that the B-spline curves in Figure 7.1 and Figure 7.2 do not extend to the first or last control points, though those endpoints are used in defining the shape of the curve:

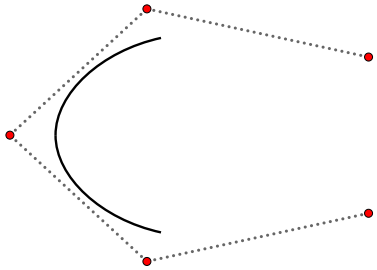


Fig. 7.2 - The first and last control points are not part of the curve

To include the first and last control points in the curve, additional (“phantom”) points are added that extend the set of control points. Figure 7.3 shows the additional phantom points (in green) and the resulting B-spline curve:

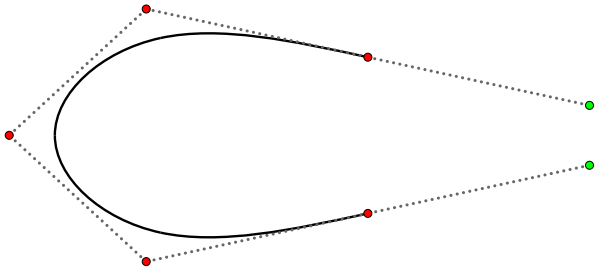


Fig. 7.3 - Adding points (in green) to extend the curve

A phantom point is constructed by reflecting through the endpoint the point that preceded it. Geometrically, this can be expressed as the addition of a vector to the endpoint, where the difference of two points produces a vector that displaces the second point to the first.

Given control points $p_1, p_2 ... p_n$, the phantom points can be defined geometrically by:

$$p_0 = p_1 + (p_1 - p_2)$$

$$p_{n+1} = p_n + (p_n - p_{n-1})$$

Algebraically, this simplifies to:

$$x_0 = 2x_1 - x_2$$

$$y_0 = 2y_1 - y_2$$

and:

$$x_{n+1} = 2x_n - x_{n-1}$$

$$y_{n+1} = 2y_n - y_{n-1}$$

Adding phantom points to the control points of Figure 7.1 (page 85) produces Figure 7.4:

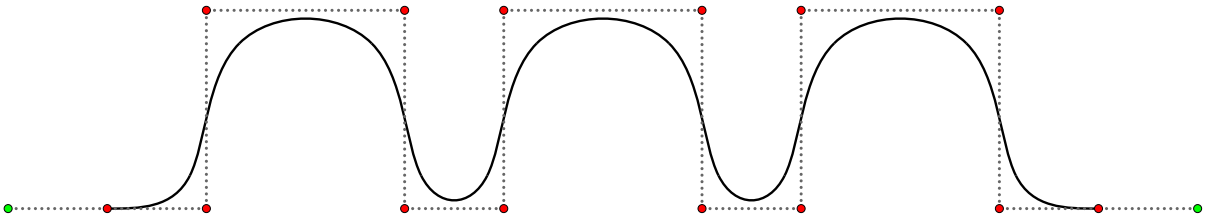


Fig. 7.4 - Extending the curve of Figure 7.1 (page 85) to the beginning and ending control points

These diagrams demonstrate the construction of phantom points in two dimensions, but the construction extends to three-dimensional control points as well.

7.6.3 The IFibers object

An instance of `IFibers` is created by a method of the `ITransaction` object:

Listing 7.1 – Creating the IFibers instance

```
IFibers *fibers = transaction->create<IFibers>("Fibers");
```

The following methods create and edit the attributes of the `IFibers` object:

`IFibers::create_attribute_vector`
: Creates an attribute for a fiber instance.

`IFibers::attach_attribute_vector`
: Attaches an attribute to the fiber instance.

`IFibers::access_attribute_vector`
: Returns a fiber attribute specified by its Mesh_attribute_name .

`IFibers::edit_attribute_vector`
: Detaches and returns a fiber attribute specified by its Mesh_attribute_name.

Currently, the only supported attribute is a texture coordinate (attribute `ATTR_TEXTURE`) defined by the first texture space. Only the first texture space is supported. Texture coordinates may be 1-, 2-, or 3-dimensional.

7.6.4 Adding a single fiber to the IFibers object

Individual fibers are added to the `IFibers` instance by the `IFibers::add_fiber` method. The method returns an instance of `Fiber_handle_struct`.

Listing 7.2 – Adding a single fiber

```
const Fiber_handle_struct fiber_handle =
    fibers->add_fiber(num_vertices_per_fiber);
```

The control points and their attributes are added to the fiber using the handle returned by `IFibers::add_fiber`:

Listing 7.3 – Adding a fiber vertex and defining its attributes

```
int vertex index;
mi::Float32_3 vertex_coordinates;
...
fibers->set_control_point(fiber_handle, vertex_index, vertex_coordinates);
fibers->set_radius(fiber_handle, vertex_index, radius);
```

7.6.5 Example - Creating fibers

Source code
`example_fibers.cpp` (page 212)

The program `example_fibers.cpp` (page 212) demonstrates the creation of an `IFibers` object with function `create_fibers_ball`. A series of single fibers are added to the `IFibers` object. Each fiber is constructed beginning with a vertex on the surface of a sphere and a series of vertices positioned increasingly farther away from it. Vertex positions are randomized to create a spiraling structure for each fiber. The resulting `IFibers` object is shaped like a ball of curly hair and is added to the scene just if it were a mesh.

The object is shaded using a multicolor material that uses the `df::chiang_hair_bsdf` BSDF defined in file `main_hair.mdl`. The MDL search path is specified by a command line argument to the program. When rendering is complete, the resulting image is written to file.

7.7 Creating particles

The following sections introduce core concepts related to the `IParticles` interface:

7.7.1 The geometry of particles

Particles are represented by a sequence of four floats, each one of these four floats specify the center of a particle and its radius. At the moment, only spherical particles can be defined.

7.7.2 The IParticles object

An instance of `IParticles` is created by a method of the `ITransaction` object:

Listing 7.4 – Creating the IParticles instance

```
IParticles *particles = transaction->create<IParticles>("Particles");
```

The following methods create and edit the attributes of the `IParticles` object:

`IParticles::create_attribute_vector`
Creates an attribute for a particle instance.

`IParticles::attach_attribute_vector`
Attaches an attribute to the particle instance.

`IParticles::access_attribute_vector`
Returns a particle attribute specified by its Mesh_attribute_name.

`IParticles::edit_attribute_vector`
Detaches and returns a particle attribute specified by its Mesh_attribute_name.

Currently, the only supported attributes are user attributes (attribute `ATTR_USER`, up to four different ones) and the motion attribute (`ATTR_MOTION`). Particles also have UV values, the same spherical mapping is applied to all the particles of a particle object.

7.7.3 Adding particles to the IParticles object

Individual particles are added to the `IParticles` instance by the `IParticles::add_particle` method.

Listing 7.5 – Adding a single particle

```
Float32 p[4];
...
particles->add_particle(p);
```

At the moment, only spherical particles can be defined so the size of the float array passed to the add_particle method must be 4 (three floats for the position and one for the radius).

Particles can also be added all at once by the method `IParticles::set_data` method.

Listing 7.6 – Adding multiple particles

```
size_t num_particles = ...;   Define number of particles
std::vector<Float32> p(num_particles * 4);
...
particles->set_data(&p.front(), num_particles);
```

Since a particle is defined by four floats, the float array passed to the `IParticles::set_data` method must contain four times as many floats as there are particles.

7.7.4 Example - Creating particles

Source code

example_particles.cpp (page 253)

The program `example_particles.cpp` demonstrates the creation of an `IParticles` object with the function `create_particles_cuboid`. The particles are first drawn randomly inside a cuboid and with a random radius, storing their position and radii in a vector of `Float32`, then they are added all at once to the created `IParticles` object, which is then added to the scene just if it were a mesh.

The object is shaded using a common diffuse material. The MDL search path is specified by a command line argument to the program. When rendering is complete, the resulting image is written to file.

8 Materials

Materials define the visual properties of objects, for example:

- The color, reflection and refraction, and light emission properties of surfaces
- The scattering and absorption properties of volumes
- Modified geometric properties of surfaces such as displacement and normal perturbation (bump mapping)

Iray provides an example catalog of physically-based materials to apply to geometry, lights, and cameras. These materials are defined using NVIDIA Material Definition Language (MDL). Materials are defined based on distribution functions for reflection and refraction, for volume scattering effects, and for light emission properties.

Material definitions define "what to compute." The "how to compute" task is considered the domain of the renderer. This split of responsibilities enables all Iray rendering modes to use the same materials and is key to constructing a unified conceptual model of rendering.

8.1 Material Definition Language

8.1.1 Overview

NVIDIA Material Definition Language (MDL) enables users to define the reflective, transmissive, emissive and volumetric properties of objects. It consists of two major parts:

1. A declarative material definition based on a powerful material model, and
2. A procedural programming language to define functions that can compute values for the parameters of the material model.

8.1.2 Additional Information

The building blocks needed for part one, like elemental distribution functions, their modifiers and combiners, the MDL file format, and the organization in modules and packages is explained in the *NVIDIA Material Definition Language: Language Specification* document [MDLSpecification] (page 197). The *NVIDIA Material Definition Language: Technical Introduction* document [MDLIntroduction] (page 197) gives an introduction.

Not all of part two is currently supported in all contexts of all NVIDIA Iray render modes. Environment functions (page 119) are fully programmable in all modes. In some other cases, expressions are restricted to a set of predefined functions and combinations of those, plus constant-folding in the MDL compiler. The available predefined functions and where and how they can be used is explained in the documentation for the `base.mdl` module. MDL state transformations used in constant expressions are ignored (the identity matrix is used instead).

The release tape contains some example MDL materials located in the directory mdl/material_examples, which showcase some of the capabilities of the material definition language and the texturing functionality exposed through the module base.mdl.

8.1.3 Integration in Iray

MDL materials and functions are used in the following places in Iray:

- MDL modules can be imported into the scene database through the `IImport_api` methods. The special variable `{shader}` can be used as prefix in URIs to resolve the relative suffix against the shader search paths. For example, the URI `{shader}/base.mdl` imports the `base` module from the shader search paths. The import creates database elements of the following classes to represent the corresponding MDL elements:
  - `IModule`
  - `IFunction_definition`

  Note: The MDL importer supports relative URIs only.

  For resources in string-based MDL modules the following adjustments to the file path resolution procedure in Section 2.2 of [MDLSpecification] (page 197) apply:
  - The *current working directory* and *current search path* are undefined.
  - The *current module path* is obtained from the fully qualified package name by replacing all "::" subtrings by slashes.
  - Normalization of absolute file paths remains unchanged.
  - Normalization of strict relative file paths always fails since the file required to exist is treated as missing.
  - Normalization of weak relative file paths behaves as if the file to test is missing; in effect, a slash is added to the front.
  - The consistency check for absolute and weak relative file paths always succeeds (the preconditions of the check do not apply because current working directory and current search path are undefined).
- A material definition can be cloned, albeit creating a new material with new default values in a module using the `IMdl_module_builder::add_variant`() method. The new default value can be function calls and thus whole material graphs can be created and stored with this method.
- MDL modules can be exported from the scene database through the `IExport_api` methods.
- A material definition can be instantiated with argument values for its material parameters, which results in a `IMaterial_instance` database element. Parameters can continue to be updated on material instances.
- A function definition can be provided with argument values for its parameters, which results in a `IFunction_call` database element. Parameters can continue to be updated on the function call. Function calls can be used in arguments to other function calls and material instances.

- The `material` attribute can be used on geometric objects to apply a full MDL material to the object and on lights to apply the emission characteristics of a MDL material to the light. The `material` attribute references a MDL material instance represented by the `IMaterial_instance` in the Iray API.
- A function call can be used to define the environment lookup function (page 119).
- A function call can be used to define the virtual backplate texture (page 131).
- The Iray API provides two wrappers `Definition_wrapper` and `Argument_editor`. The first wrapper simplifies usage of the interface `IFunction_definition`. The second wrapper simplifies usage of the interface `IFunction_call`.

8.1.4 Hair Materials and Fiber Geometry

Starting with MDL 1.5, MDL materials may specify a hair BSDF (`material.hair`) to describe reflection and transmission characteristics of cylindrical fibers, such as hair and fur. If a non-default constructed hair BSDF is present in the material, Iray uses that on fiber geometry for shading. On non-fiber geometry the hair BSDF is also supported, but note that in absence of actual local cylinder geometry the shading results may be inconsistent with expectations. Fiber geometry in all cases is the preferred solution to use the hair BSDF, triangulated geometry should only be used as a last resort (e.g. on legacy scenes/assets). In case no hair BSDF is specified in the material, the regular BSDF is used on fiber geometry, too. Note that for regular BSDF shading fiber geometry is always single sided (so only `material.surface` is used) and opaque (i.e. transmission is discarded).

On fiber geometry, the following state is provided to MDL functions as `state::texture_coordinate(index)`:

- The first texture space (index 0) provides a normalized texture coordinate from 0 at the fiber start to 1 at the fiber end in the x-component, a normalized texture coordinate around the fiber in y-component, and the fiber radius in world space in the z-component.
- The second texture space (index 1) contains per fiber texture coordinates as specified for the `IFibers` object. If none are provided, it provides the coordinates of the fiber start point in world space by default.
- The third texture space (index 2) contains per vertex texture coordinates as specified for the `IFibers` object.

8.1.5 Bump mapping energy loss compensation

`string iray_diffuse_bump_energy_compensation = all`

Differences of shading and geometric normal cause energy loss for diffuse BSDFs as the actual surface geometry then shadows parts of the shading hemisphere. This is most pronounced for strong bump/normal mapping, where up to half of all energy is lost. Iray by default compensates that energy loss. To restore the old legacy behavior one can use `"none"` (no compensation), `"brdf"` (only diffuse reflection is compensated) and `"btdf"` (only diffuse transmission is compensated). By default `"all"` (compensate both) is used.

8.2 Measured materials

Iray supports the use of measured material data as building blocks in the material model of the "Material Definition Language" (page 90). The more common scenarios for measured materials are:

- Measured isotropic BSDF (page 93)
- Spatially varying BSDF (page 93)
- Measured reflection curve (page 94)
- Measured light profile (page 94)

Note that the measured data does not necessarily have to come from real measurements. On the contrary, these representations can be very useful for working with synthesized data from models that cannot be represented otherwise in MDL, such as a new BSDF model.

8.2.1 Measured isotropic BSDF

Measured isotropic BSDF data can represent measured reflection data (BRDF) as well as measured transmission data (BTDF). Both are represented in a single file format, the *MBSDF file format* documented in the appendix of the *NVIDIA Material Definition Language: Language Specification* [MDLSpecification] (page 197). A single file can thus describe the full scattering characteristics of a surface point.

The Iray API can load MBSDF files and represent them in the API with the `IBsdf_measurement` class. This class allows the data to be edited and allows new measurement data to be created in memory without a file format requirement. Values of this class can be passed to suitable parameters of MDL materials.

In an MDL material, the measured data is represented with the `bsdf_measurement` type. It is used as a parameter type on the `df::measured_bsdf` type, which is an elemental BSDF that can be used like any other BSDF type in MDL.

The `df::measured_bsdf` type has a mode parameter to select which part of the measurement — reflection, transmission, or both — is used. This parameter takes precedence over the actually available data, which allows the testing of reflection and transmission contributions independently. If this mode parameter asks for data that is not in the measurement, an error results.

8.2.2 Spatially varying BSDF

To create spatial variety, you can use a parametric surface model and drive the parameters from textures mapped to the surface. This representation has also been explored in the measured materials context, where a measurement process fits a measurement to a chosen material model and provides you with a set of matching textures to be used with that model. Work in this area usually refers to this representation as a spatially varying BRDF, or SVBRDF for short.

The flexibility of MDL allows you to create matching or closely matching parameterized material models, for example, from a simple DGS model to a multi-lobed glossy car paint with flakes. The measured textures can then be used to drive the resulting parameters. It might be necessary to pre-transform those textures in the application to match the material

model chosen, or it might be possible to transform them using the texture pipeline functions in the base.mdl module.

8.2.3 Measured reflection curve

If the conventional Fresnel effect or the custom curve provided with the MDL material model is not sufficient, you can also measure the reflection behavior of a surface and use that data. The data is represented in MDL as an array of color measurements at equidistant angles from zero to ninety degrees from the surface normal. There is no dedicated file format for this data.

Two building blocks in MDL work with a measured refection curve:

`measured_curve_factor`
: Attenuates a single underlying BSDF based on the measured reflectance curve

`measured_curve_layer`
: Layers a BSDF over a base BSDF based on the measured reflectance curve

8.2.4 Measured light profile

Light profiles represent far-field measurements of light source emissions. As a standard file format, IES light profiles are supported in Iray.

The Iray API can load light profile files and represent them in the API by using the `ILightprofile` class. This class only allows access to the data. There is no support to edit the data or create new measurement data in memory. Values of this class can be passed to suitable parameters of MDL materials.

In an MDL material, the light profile is represented by the `light_profile` type. It is used as a parameter type of the `df::measured_edf` type, which is an elemental EDF that can be used like any other EDF type in MDL.

See also "Sources of light" (page 113).

8.3 Editing and creating materials

The following sections introduce typical MDL-related tasks such as modifying inputs of material instances, creating new material instances, and attaching function calls to material inputs:

- Modifying inputs of existing material instances (page 94)
- Creating new material instances (page 95)
- Attaching function calls to material inputs (page 95)

The program `example_mdl.cpp` (page 230), which uses the scene `main.mi` to demonstrate typical MDL-related tasks.

8.3.1 Modifying inputs of existing material instances

This section describes the steps for modifying the input of an existing material instance. In particular, it explains how to change the color of the material instance `yellow_material`, which is used for the cube and other scene elements in `example_mdl.cpp` (page 230).

To change the color, we retrieve the current argument expression and clone it to obtain a mutable instance. Here, the argument expression is a constant and the get_value() method gives access to its value. Alternatively, we could have created the value and the corresponding expression from scratch as shown in the section below.

To set the value, the free function `set_value` is used. Alternatively, we could have retrieved the interface `IValue_color` and used its set_value() method directly.

The new value is then simply set with the set_argument() method.

These steps are shown here in full detail for illustrative purposes. The helper class `Argument_editor` can be used to simplify such workflows, as shown in the second code block.

8.3.2 Creating new material instances

This section describes the steps for creating a new material instance and an example for its use.

As described in the previous section, the material instance of `yellow_material` could be shared by a number of scene elements. When the color of this material instance is changed, then the change is reflected in all scene elements that use the shared material. To change the color of the cube only, requires three steps: preparing the arguments of a new material instance, creating the new material instance, and then attaching it to the cube. The steps are described in more detail as follows:

1. Prepare an expression list that holds the arguments of the new material instance. This is mandatory if the material definition has parameters without default initializers, otherwise it is optional. In both cases, arguments can be modified after creation of the material instance as shown above. Here, we create the new argument expression from scratch.
2. Use the method create_material_instance() to create a new material instance from the material definition mdl::main::diffuse_material (the same definition that was used for yellow_material). The newly created material instance is stored in the database.
3. Replace the previously used material instance, `yellow_material`, with the newly created material instance by changing the `material` attribute of cube_instance.

These steps are shown in `example_mdl.cpp` (page 230) in full detail.

8.3.3 Attaching function calls to material inputs

The preceding sections of this section focused on constant inputs for material instances. This section describes how to connect another MDL function to the input of the material instance.

All arguments of material instances or function calls are expressions. There are two important types of expressions: *constants* (represented by `IExpression_constant`) and *calls* (represented by `IExpression_call`). In the previous two sections constants represented a color. Now a call expression will attach a different MDL function to define the color.

The program `example_mdl.cpp` (page 230) shows how to apply a texture to the ground plane:

1. Create the necessary database elements of type `IImage` and `ITexture`, which will hold the texture to be used.

2. Import the MDL module `base`, which contains the MDL function mdl::base::file_texture to be used.
3. Instantiate the function definition mdl::base::file_texture with the argument `texture` set to the database element of type `ITexture` created in step 1. Actually, the true name of the database element representing the MDL function base::file_texture is not mdl::base::file_texture, but quite a long name that includes the function signature. Hence, the method `IModule::get_function_overloads` is used to retrieve the exact name of the database element without hard-coding it.
4. The return type of mdl::base::file_texture is the structure `texture_return`, which does not match the desired argument type `IType_color` of the `tint` argument of the material instance. Hence, the MDL function mdl::base::texture_return.tint is used, which extracts the `tint` field of the `texture_return` structure. In this case, the helper function get_function_overloads() was not used, rather the full name mdl::base::texture_return.tint(::base::texture_return) directly.
5. The function definition is instantiated and its `s` argument is set to the instance of mdl::base::file_texture.
6. The `tint` argument of the material instance used for the ground plane is changed to point to the new function call. The program `example_mdl.cpp` (page 230) shows how this step can be simplified using the helper class `Argument_editor`.

8.3.4 Example – Using MDL materials and functions

Source code

`example_mdl.cpp` (page 230)

This example imports a scene file, executes various MDL operations on the scene, and writes the rendered image to a file.

9 Decals

Decals are a first-class scene database element that allows you to place decals, such as stickers and labels, on objects in the scene, give them full MDL materials on their own, place them in space and selectively enable or disable them. The placement of decals on objects is physically based insofar as they can be placed on either side of a surface, they can overlap each other, and you can see through transparent decals and see other decals below it.

9.1 Decal scene element

The decal scene element is represented through the `IDecal` interface. The implicit geometry of a decal is a unit square centered at the origin in the x-y-plane facing positive z-direction.

A decal scene element has the following fields and attributes:

- A clipping box that restricts the effects of the decal. The clipping box is an axis-aligned box defined in the object space of the decal. Note that the clipping box is different from a bounding box here in that it is part of the modeling capability with decals and not a conservative enclosure.
- A projector that defines how the decal is applied to the object. Technically, a projector returns for the current surface position the uv-coordinates that have to be used for the material on the decal.

  A projector is thus a MDL function in Iray. The supported MDL function is `base::coordinate_projection()` with its `coordinate_system` parameter left at its default value of `texture_coordinate_object`. Without a projector function, the decal will use the uv-coordinates of the object itself, which is the default. This makes the decal behave more like texturing but with a full material layered on top of the objects material.

  With or without projector, in both cases a base::transform_coordinate() function can be used in addition to transform the resulting uv-coordinates and thus position the decal in texture space.
- A texture space index to be used by the projector to temporarily store the uv-coordinates for the decal material evaluation. The default texture space is 0.
- A signed integer priority value which is used to resolve the ordering in case of overlapping decals. The default is 0 and negative values are allowed. A decal with higher priority is placed on top of a decal with lower priority.
- A `material` attribute (not a field). This attribute is the same as for any other object, except that decals support only a single material and not arrays of materials.

Materials on decals have a few restrictions:

- Materials are always thin-walled on decals.
- Front and back-side can have different surface properties. For example a label on a glass bottle can have the logo printed on the front-side while the back-side has a plain white paper surface.
- Decals support cutout opacity and transmissive materials.
- Emission is not supported.
- Normal maps are supported.
- Displacements are not supported.
- Volume properties are not supported, except that the IOR value is used in valid thin-walled surface material effects, like in the Fresnel-driven selection between reflection and transmission in the combined specular BSDF.

- An enum of type `Decal_face_mode` that controls whether the decal appears on the front face (the default), the back face, or both faces of the objects surface. The face orientation is determined w.r.t. the projection direction. The normal derived from the orientation of the geometric primitives is not relevant. For non-thin-walled materials the decal is never shown on the face with higher density.

9.2 Placement of decals in space

Decals are placed in space similar to geometry objects with instance transformations using the `IInstance` nodes.

However, unlike geometry objects, decals are not independent scene objects and they (or instances of them) are not connected in the scene graph using groups. Instead, decals or instances of decals are attached to the scene graph with the `decals` attribute, for example, on the geometry nodes themselves, on instance nodes or on group nodes.

The transformation between world space of the scene graph root group and the local object space of the decal consist of the concatenation of the individual instance transformations from the root group down to the node with the `decals` attribute and from there to the decal.

Decals can be instanced multiple times much like geometry objects, for example by using a decal in several instances, or attaching it to a node that is instanced multiple times. In addition, a decal or instances of a decal can be attached multiple times at different nodes, or even at the same node, for example, with different instance transformations.

This mechanism of placing decals in the scene and positioning them can be used to achieve different dynamic behaviors on scene updates. For example, given a car modeled with a car body and a movable door, a single decal can be used to place a sticker across the door and car body. When the door opens, the following two interesting scenarios can be modeled as detailed below:

1. The conventional behavior of decals would leave the decal stationary, so that the decal keeps projecting, now distorted, onto the door.
2. The part of the decal projecting onto the door rotates with the door and behaves like a sticker glued to the car.

The following two figures illustrate the scene graphs that model these two scenarios. In both, the transformation T2 is used to rotate the door.

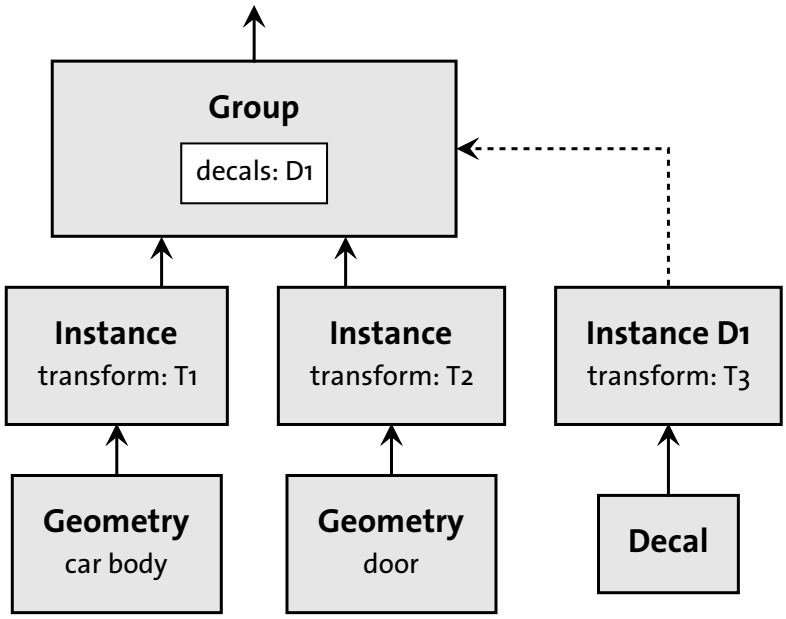


Fig. 9.1 - The geometry is modeled conventionally with instances connecting to a group.

In Figure 9.1, attaching the decal with its own instance to the same group keeps all involved transformations, T1, T2 and T3, independent from each other.

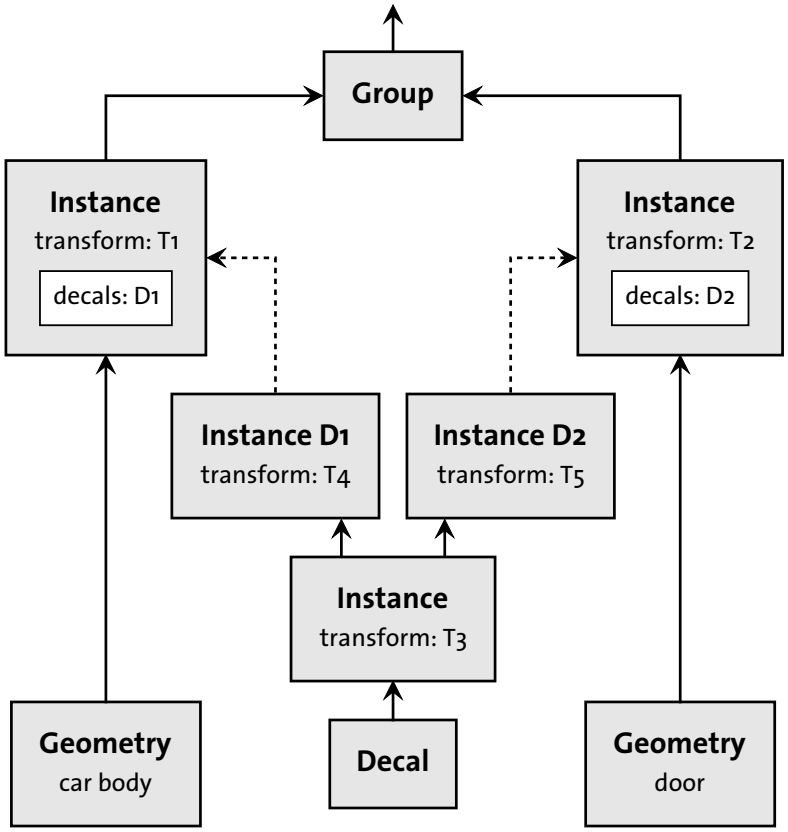


Fig. 9.2 - The geometry is modeled identically to Figure 1, but the decal is now attached to the geometry instances.

In Figure 9.2 (page 100), attaching the decal to the geometry instances effectively instances the decal twice. This makes the decal dependent on the geometry transformations, T1 and T2. In addition to the single instance transform T3, which transforms and positions the decal in one piece, this example adds two more instance transforms, T4 and T5. These two transforms independently place the decal on the car body and the door. For example, an initial setup would initialize T4 to the inverse of T1 and T5 to the inverse of T2, resulting in the same initial placement of the decal as in the modeling of Figure 9.1 (page 99).

9.3 Layering and placement of decals on objects

Decals can overlap each other and they can be visible on the front side or the back side of a surface.

When overlapping, decals create physical layers that determine the visibility of the decals. Transparency and cutout-opacity is also supported in that a transparent material of a decal allows you to see the decals below.

The order of overlapping decals is determined by their priority. A decal with a priority greater than another decal is above the other decal. If they are of equal priority, the decal that comes first in the sequence of decals in the `decals` attribute is above the other decal.

You can use the enum field value of type `Decal_face_mode` to control on which side of a surface the decal is visible, the front side, the back side, or both sides. The front side of a surface for the purpose of decal visibility is the side whose angle between the implied face normal and the projector direction is larger than 90 degrees. The other side is the back side. If there is no projector then the front side is defined by the orientation of the surface primitives.

In addition, for surfaces with a non-thin-walled base material, a decal is never shown on the side with the higher density, determined by the side with the higher index-of-refraction value. For example, this difference will be relevant for a label on a glass bottle using a cylindrical projector when the label should only be seen on the outside of the bottle However, a label on the inside of the bottle can be made visible with the `DECAL_ON_BACK_FACE` value for the `Decal_face_mode` enum value.

9.4 Selectively enabling and disabling decals in the scene graph

By default, all decals attached with the `decals` attribute are visible on all objects below the node where the attribute is attached.

The additional attributes, `enabled_decals` and `disabled_decals`, give you more fine-grained control over which decals are visible on which object. To understand their behavior, attribute inheritance needs to be understood for these attributes.

The inheritance of the `enabled_decals` and `disabled_decals` behaves like all other attributes. If they are not set explicitly, `enabled_decals` will be set to the value in `decals` by default, and `disabled_decals` will be empty by default, effectively making all decals visible under the rules explained below.

The inheritance of the `decals` attribute is special: the array elements of the child are always prepended to the array elements of the parent. Note that inheritance does not change the behavior of transformations explained above.

All three attributes are inherited down the scene graph and end in a geometry leaf node. A decal is now visible on a geometry leaf node if and only if it is listed in the `decals` attribute and in the `enabled_decals` attribute but not in the `disabled_decals` attribute.

The interplay of these rules can lead too fairly complicated constellations. We thus recommend to look first at a simple policy and see if it suffices to realize the desired modeling and workflow needs. The following example can serve as one.

One policy could be to attach decals at the root group — they will not move with the geometry then — and select their visibility explicitly at the instance nodes referencing the geometry. The example scene graph shown in Figure 3 illustrates this by modeling the tires of a car with four instances of the same tire geometry. The two decals in the scene graph are attached to the root group and each instance defines the `enabled_decals` attribute to define all the decals that are visible for this tire. In this example, decal D1 is visible on all tires and decal D2 only on the fourth tire.

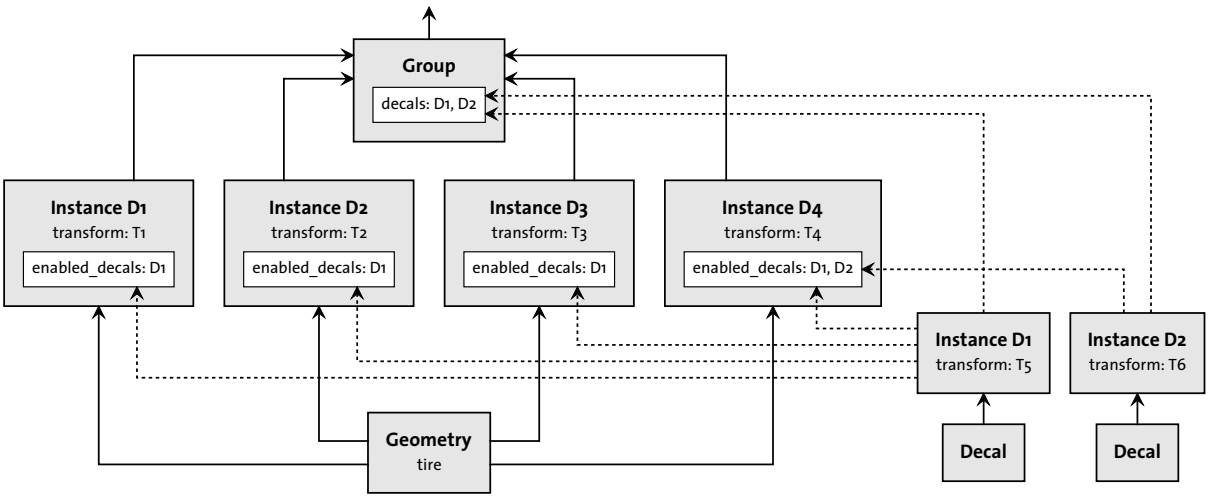


Fig. 9.3 - A scene graph modeling four tires of a car with instancing and illustrating the use of the root group

The placement of all decals in the root group implied that they will not move with the geometry. For the decal D1 that is visible on all four tires, this also means that it is not positioned by the instance transformtions to be at the same place on all four tires. To model that, the decal would need to be attached to the four instance nodes.

10 Inhomogeneous volumes

Inhomogeneous volume support provides the ability to render effects like clouds, smoke or fire, but can also add structure to subsurface scattering materials. For the first use case there exists a special scene element type, while for the latter only a change to an existing MDL material is required. This chapter covers the setup for both application cases.

10.1 Volumetric textures

Since both use cases require a volume definition, Listing 10.1 is an example of such a material:

Listing 10.1

```
export material inhomogeneous_volume(
    uniform texture_3d density = texture_3d("/smoke.vdb"),
    uniform color albedo = color(0.5f),
    uniform float4x4 transform = float4x4(1.0f),
    uniform float density_multiplier = 1.0f,
    uniform bool density_relative_to_size = false,
    uniform color ior = color(1.0f))
= let {
    base::volume_coefficients c = base::lookup_volume_coefficients(
        density, albedo, 1-albedo, color(0), color(0), transform,
        density_multiplier, density_relative_to_size);
} in material(
    surface: material_surface(
        scattering: df::specular_bsdf(mode: df::scatter_reflect_transmit)),
    ior: ior,
    volume: material_volume(
        scattering_coefficient: c.scattering_coefficient,
        absorption_coefficient: c.absorption_coefficient)
);
```

The most important property is the 3d texture `density` which loads the OpenVDB[1] dataset named `smoke.vdb` for this example. Only VDB files are supported as valid input textures. The VDB format specifies inhomogeneous data on an integer grid, along with an internal transform of the raw volume data which places this data into world space. The function `base::lookup_volume_coefficients` takes a sample in object space, but if the instance transformation of the object is an identity matrix, this directly reflects to the world space. The region in space for which volumetric data exists will be referred to as the *VDB domain* in the following sections.

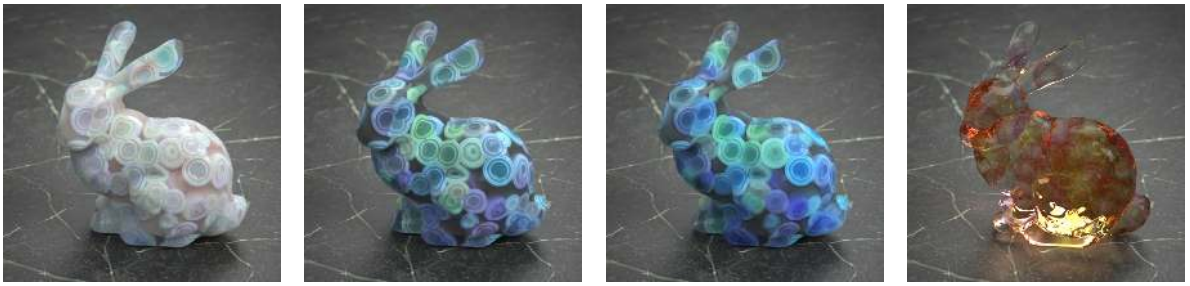
1. `https://www.openvdb.org/`

A VDB file may contain more than one data channel. In this case a specific channel can be selected as specified by MDL.

10.1.1 Colors

Depending on the value type of the selected channel, the data is either interpreted as scalar (one or two values per texel) or as tristimulus color (three or four values per texel). Note that in case of two or four values existing per texel, the last component is ignored. When using a tristimulus color channel as input, the data is assumed to be in the rendering color space without conversion. However, if rendering in spectral mode ("Spectral rendering" (page 52)), these input values are converted into a spectral representation on the fly using the method described in *Ray Tracing Gems 2* [Jendersie21] (page 197).

The `base::lookup_volume_coefficients` function takes four colors as input. For both scattering and absorption, a multiplier and an offset enable a single inhomogeneous value to be read from the VDB file and used in various ways. Figure 10.1 show different possible configurations. Note that negative numbers are allowed.



*Constant albedo: Scattering=0.9*D+0, Absorption=0.1*D+0*

*Constant absorption: Scattering=1*D+0, Absorption=0*D+1*

*Variable albedo: Scattering=1*D+0, Absorption=-1*D+1*

*Variable absorption: Scattering=0*D+0, Absorption=1*D-0.1*

Fig. 10.1 - Example usage of volume parameters to achieve different effects with a single inhomogeneous data source "D". The numbers are read "color(x)" for the respective multiplier or offset.

10.1.2 Density relative to size

Fig. 10.2 - Volume at original size (left), a scaled volume with density scale set to `true` *(middle) and without scaling (right).*

The last parameter `density_relative_to_size` of `base::lookup_volume_coefficients` handles how the instance transform affects the volume's density. If enabled, a reduction in size will cause an increase of the density and vice versa, such that the optical appearance of

the object is preserved. A scaled object with relative scaling enabled will have a similar appearance to a scaled image of the original volume.

Otherwise, when setting this option to `false`, the physical density of the medium will be preserved. A scaled object will therefore become more or less opaque, as seen on the right in the image. Disabling density scaling is to be preferred if, for example, the physical medium is defined once and used in several object instances to increase scene variation.

10.1.3 Interpolation

Data values in the volume are interpolated using a trilinear texture filter.

10.2 Inhomogeneous volumes as subsurface materials

Fig. 10.3 - Procedural inhomogeneous volume applied as a subsurface scattering texture

A material like `inhomogeneous_volume` in Listing 10.1 (page 103) can be attached to any object as usual. Subsurface scattering effects within the object will then feature varying density.

The object itself defines the boundary hull for *all* volume lookups. Materials within mesh bound objects are always subject to the usual volume stack. If a mesh object overlaps with a volume object (page 106), only the material of the mesh object will be used on the inside. If two mesh objects overlap, one of the two subsurface materials is used exclusively depending on the intersection order of the surfaces.

Note that when attaching a material to an object, and the object's boundary is outside of the domain of the volumetric data, the object appears to be empty. It is still possible to see the boundary depending on the surface material, but there would not be any volumetric interaction on the inside. To match object boundary and volume domain, `transform` can be used: Setting it to map the VDB domain into the bounding box of the object is a useful starting point.

Transformations to the object also apply to the volume. Translation or rotation do not change the object's appearance while scaling may cause density changes (see "Density relative to size" (page 104)).

The `ior` and `surface` properties of the material are respected by the mesh boundary. This can be used to render, for example, a cloud in a glass ball or similar effects.

10.3 Volume objects

Fig. 10.4 - A cloud rendered using a volume object.

Volume objects are another scene element opposed to meshes. They are explicitly designed for volumetric effects like fire, smoke, clouds or fog and have numerous advantages over the subsurface scattering from the previous section.

- Overlaps are handled additive in density. You can even mix volumes with different colored particles or scattering functions smoothly. Doing the same with a mesh boundary would fallback to the material stack and either material would be used exclusively in the overlapping region.
- Illumination of volumes from small lights sources will have significantly less noise.
- Illumination on other scene objects from emissive volumes will have significantly less noise.
- Having lights or the camera partially inside the volume and partially outside is possible.
- Support of clipping planes works better, because volume objects can be entered or left through a clipping plane reliable. While the inner medium of a mesh might be detected as well, the latter can fail as it depends on normal orientation. Especially, other objects embedded inside the volume can make problems in mesh-based volumes.

The key difference is that volume objects are not restricted by explicit boundary hulls. Instead, these are clipped to the VDB domain and all surface properties, as well as the index of refraction, are ignored for volume objects. Using `transform` in the MDL material or the volumes' `bbox` selects a subregion of the volume, which is clipped to the VDB domain boundary.

Other, potentially transparent or refracting, objects can be embedded inside volumes. The volume will be clipped on the inside of the embedded objects. That means inside any object its subsurface scattering properties will be used and not that of the surrounding volume. This does not apply to thin-walled objects because they do not have an inside by definition. These

clipping objects may be fully transparent to enforce some cipping of the volume only. As another example, clipping also enables scene modeling that prevents volumetric data from leaking into the interior of objects, like an airplane flying through clouds. However, the airplane must consist of a watertight mesh to avoid the appearance of clouds on the inside.

10.3.1 Example - Creating a volume object

The volume creation process is similar to the creation of other scene elements. However, special care must be taken when setting the volume object bounding box and transform. The volume's bounds define where clipping will take place. If these bounds are outside the VDB domain then nothing will be visible, but if the bounds only cover a small fraction of the domain, it will clip the volume accordingly. A reduction of the bounding box will cause a clipping in grid space, while a change of the transformation matrix will clip in world space. Keep in mind that both parameters form an invisible, oriented box as the boundary of the volume. The actual data stays at the same position and orientation — only the window to that data will be changed. To transform the volume as a whole, the transformation of the instance can be used.

Note that it is also possible to achieve a clipping effect by using the UVW transform parameter within `base::lookup_volume_coefficients`. In effect, this moves the data without compromising the volume object boundary or the mesh boundary.

A code example that loads an OpenVDB file and sets the proper bounds and transformations is given here:

Source code
`example_volumes.cpp` (page 325)

10.4 Emission

Fig. 10.5 - A fire using inhomogeneous blackbody emission.

A volumetric medium may emit light on its own. Prominent examples are flames and gas-discharge lamps. To describe those, you can assign a value to the `emission_intensity` of MDL volume materials. For example, Figure 10.5 (page 107) shows an MDL material assigned to the geometric structure typical of flames.[2]

In the case of uniformly glowing bodies a simple constant color suffices. The emission of flames is more involved as it requires proper inhomogeneous data or procedural functions. Most tools that simulate explosions and fires provide this in form of a temperature field in a VDB file that is then used to generate blackbody emission. The MDL base function `volume_blackbody_emission` simplifies the creation of such objects:

Listing 10.2

```
color volume_blackbody_emission(
    uniform texture_3d temperature,          Temperature in arbitrary units
    uniform texture_3d density = texture_3d(),   Density of emissive particles
    uniform float temperature_mult = 1.0f,   For custom units, use temperature_mult to convert the temperature texture values to Kelvin
    uniform float temperature_offset = 0.0f, For custom units, use temperature_offset to convert the temperature texture values to Kelvin
    uniform float scale = 1.0f,              Scale final emission
    uniform color tint = color(1,1,1)        Arbitrary color modification
);
```

The first and most important input is the temperature field that is provided together with the usual density in appropriate VDB files. In the above API example, a field selector must be passed to the creation of the volume texture, as in Listing 10.3:

Listing 10.3

```
int result = volume_data->reset_file("fire.vdb", "temperature");
```

Since these files are sometimes in meaningless or incorrect physical units, the `temperature_mult` and `temperature_offset` here can convert the sample values from the texture to degrees Kelvin. For example, if the input file provides the temperature in Celsius, an offset of 273.15 can be used to convert to the Kelvin scale. Below roughly 800K there will be no visible emission; for textures with such values, the multiplier and offset should be increased.

2. Scene assets used under the Creative Commons license:
https://creativecommons.org/share-your-work/public-domain/cc0/
Fire: https://jangafx.com/software/embergen/download/free-vdb-animations/

The density input is used to describe the amount of hot particles (opposed to their temperature). A larger density directly transforms into a brighter emission, but usually the absorption of the medium should increase in this case as well. If no input is supplied, a uniform density of one is assumed. This might already be satisfying, but is usually not the proper model of fire. If no explicit density of emissive particles is given, consider using the same density input that is used for the other volume coefficients.

Finally, `scale` and `tint` can be used to adjust the final brightness and to modify the color. Listing 10.4 shows an example MDL shader that was used for the introducing image

Listing 10.4

```
export material blackbody_volume(
    uniform texture_3d density = texture_3d(),
    uniform texture_3d temperature = texture_3d(),
    uniform color albedo = color(0.7, 0.7, 0.7),
    uniform float density_multiplier = 1.0,
    uniform float anisotrophy = 0.0,
    uniform float temperature_offset = 0.0f,
    uniform float temperature_mult = 1.0f,
    uniform float emission_scale = 1.0f,
    uniform color emission_tint = color(1.0f, 1.0f, 1.0f)
)
= let {
    float save_density_mult = math::clamp(density_multiplier, -1e16, 1e16);
    base::volume_coefficients c = base::lookup_volume_coefficients(
        density, albedo, (1-albedo), color(0.0f), color(0.0f),
        float4x4(1.0f), save_density_mult, false);
} in material(
    surface: material_surface(
        scattering: df::specular_bsdf(mode: df::scatter_reflect_transmit)),
    ior: color(1.0),
    volume: material_volume(
        scattering_coefficient: c.scattering_coefficient,
        absorption_coefficient: c.absorption_coefficient,
        emission_intensity: base::volume_blackbody_emission(
            temperature,
            density,
            temperature_mult, temperature_offset,
            emission_scale * density_multiplier, emission_tint
        ),
        scattering: df::anisotropic_vdf(
            directional_bias: anisotrophy
        )
    )
);
```

Note that you are not limited to the use of `volume_blackbody_emission`. If your input VDB contains final emission, you can fetch the texture more directly within the shader.

10.5 Restrictions

Though there is no direct limit for the maximum number of volumes, you should only use a small number of volume instances. Where feasible, multiple volumes should be combined into a single VDB file to improve performance.[3] In general, any number of volumes takes considerably more time than surfaces, usually with more initial noise.

- There is no repeat or clamp handling at the boundaries of a volume. Every position outside the volume will feature zero density.
- Using at most one `base::lookup_volume_coefficients` per material with all its parameters being deducible at compile time is much faster than other configurations. The same is true for `base::volume_blackbody_emission` where both can be used together like in the example above. Any custom function on any of the volume properties will trigger the much slower code path.
- The voxel format must be either float or float3 using the standard grid layout in VDB.
- Trilinear interpolation is the only interpolation method.

10.6 Atmospheric ground fog

Fig. 10.6 - Ground fog applied to Albert Mansion. (Scene courtesy of Daz 3D.)

In Iray, *ground fog* is implemented with a simple built-in volume type to place a height dependent mist into a scene. Above a threshold height the volume's density decays exponentially until it vanishes. Below that height the volume has a uniform density.

The fog must be enabled by setting `iray_ground_fog` to `true`. Following parameters define the appearance of ground fog:

3. https://www.openvdb.org/documentation/doxygen/codeExamples.html#sCombiningGrids

`iray_ground_fog_albedo`
A color in [0.0,1.0] to modify the absorption of fog particles.

`iray_ground_fog_height0`
Ground level reference height in meters. Below that height the volume has density `iray_ground_fog_density0`; above that height it decays exponentially.

`iray_ground_fog_density0`
A colored extinction coefficient for the fog density on the ground level. The value is a positive unbound density of particles per meter. Higher densities are more opaque.

`iray_ground_fog_height1`
A second height in meters which must be larger than `iray_ground_fog_height0`.

`iray_ground_fog_density1`
A colored extinction coefficient for the fog density that is reached at `iray_ground_fog_height1`. Above that height the fog continues to decrease in density. The value must be smaller or equal density0 and will be clamped otherwise. If both values are equal in any component, the fog becomes uniform. Note that the uniform fog still extends to infinity and will thus block all light from infinite environments.

`iray_ground_fog_scattering_type`
Sets the volume distribution function for the particles in this volume. Can be either "hg" or "approx_mie" which map to `df::anisotropic_vdf` and `df::fog_vdf` in MDL respectively.

`iray_ground_fog_anisotropy`
Scattering anisotropy of the Henyey-Greenstein phase function in [-1.0,1.0]. A value of -1.0 results in (specular) back scattering, 0.0 is isotropic scattering and 1.0 is (specular) forward scattering. This is only used when the scattering type is "hg".

`iray_ground_fog_particle_size`
When the scattering type is "approx_mie" this sets the water droplet diameter in micro meters. Common particles sizes for fog and clouds go from 5 to 16 micro meters. At zero micro meters this becomes Rayleigh scattering.

All parameters together define a slab where the density at the lower and the upper boundary are explicitly controlled. The exponential decay is derived from these values by:

```
decay = (height0 - height1) / log(density1 / density0)
```

The `decay` value is used to obtain the final extinction coefficient `sigma_t` by:

```
sigma_t(height) = density0 * exp(-max(0, height - height0) / decay)
```

10.6.1 Best practice

Setting the parameters of the ground fog often behaves unintuitively for humans due to its nonlinear behavior and the fact that it extends unbounded in both directions. It is possible to drop the upper density on the user side and instead compute a relative value based on the lower density. The smaller that value, the less fog will be visible at the upper high, but it will also push down the perceived heigh a lot. A factor of `1/1024` makes the fog very thin at the top, although still noticeable, and works quite well. It essentially results in a halved density at `10%` between the two heights, a fourth of the density at `20%` and so on.

The second hard to steer parameter is the remaining `density0`. Setting the extinction gives us direct control over the physical property but is relatively meaningless to us. A possible alternative is to define an expected free flight distance by:

```
density0 = 1 / ref_distance
```

where `ref_distance` is the new user parameter that specifies the expected distance at which a fog particle will be hit. Instead of the constant 1 it is also possible to use `log(2)` in which case the ref_distance would specify the distance at which 50

A further source of confusion can be the color. Often, selecting a color with a color picker results in an unexpected result of the inverse color. There are two parameters that influence the behavior with respect to color that can be changed individually or at the same time. First, the `iray_ground_fog_albedo` sets the color of fog particles. As long as the extinction is gray valued this means that any light that is not scattered is only darkened over distance. Only inscattered light will be tinted by the albedo or a multiple thereof, if scattered several times. Second, both `iray_ground_fog_density` values can be used to change the color of transmitted light. A higher value in one color channel means that the light is more likely to hit a particle in that particular channel. A light source seen through the fog will show the inverse color of what is defined, but inscattering will add more light of the respective color assuming a gray valued albedo. This is useful to model aerial perspective with the ground fog. Our atmosphere scatters blue light much stronger than green or red light. For example using a color of `(0.1,0.2,0.4)` with different scales for the two densities and an albedo close to one produces this effect.

11 Sources of light

Traditional computer graphics definitions of the source of light in a scene are based on simplifications of position, direction and a simple model of the directed illumination provided by spotlights. Later light source models included geometric extent from which light was emitted, called *area lights*. As a special case of area lights, a sphere of infinite radius around a point, the *environment*, defined a mapping of position to light intensity, represented either as a texture map or as a function returning intensity values.

To these traditional lighting concepts, Iray adds the ability to define any surface as *emissive* through its *material definition* defined by the "Material Definition Language" (page 90) (MDL). This definition applies the idea of a distribution function to light emission. The components of the definition describe how light is emitted at a single point on the surface. The combined emissive effect of all points defines the emissive behavior of the surface as a whole.

11.1 Concepts of light representation in Iray

For a single point on an emissive surface, all light energy will be emitted into the hemisphere defined by the surface normal at that point. In Figure 11.1, this is represented in a simplified form in two dimensions (like a slice of the real three-dimensional scene). A single direction within this hemisphere represents the fundamental concept in MDL's light emission model.

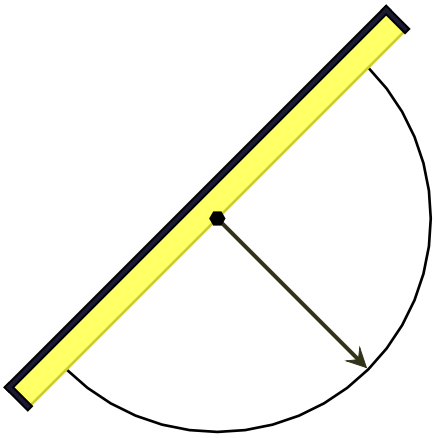


Fig. 11.1 - The hemisphere over a point on an emissive surface

Light from a point on an emissive surface that is emitted evenly in all directions is called *diffuse emission* (by analogy to diffuse reflection). In Figure 11.2 (page 114), the evenly distributed illumination of diffuse emission is represented by arrows that are the same length in all directions.

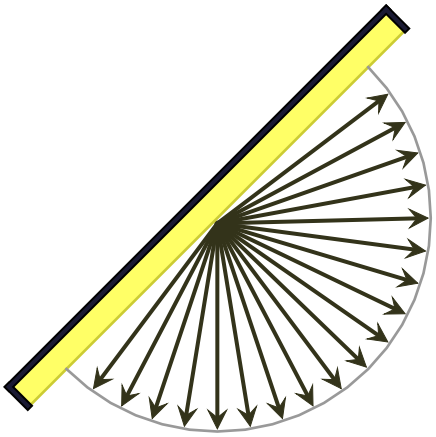


Fig. 11.2 - Diffuse emission from a point on a surface

In many emissive objects in the world, however, light will be emitted unevenly in different directions. MDL defines the way that light is distributed from a single point on an emissive surface by its *emission distribution function* (EDF). In Figure 11.3, the length of each arrow represents the amount of light emitted in that direction.

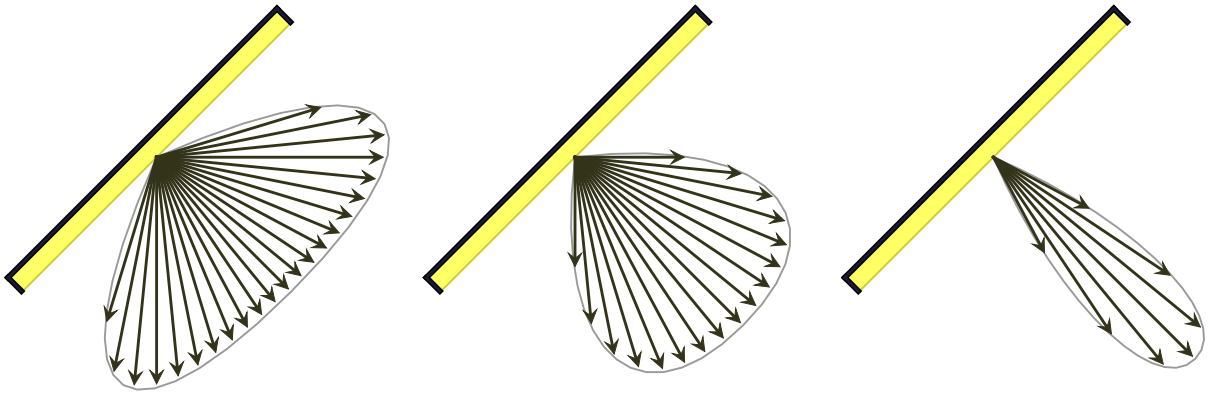


Fig. 11.3 - Variations of the emission distribution function (EDF)

The emission distribution function only defines the distribution properties of a single point on the surface. The sum of the emissive values of all points (the integral over the surface) defines the total emission of the surface itself. This calculation is done by Iray; the specification of emission by the user of MDL is through the definition of the EDF and the *radiant exitance*, a measure of how much energy is available for distribution into the hemisphere at each point.

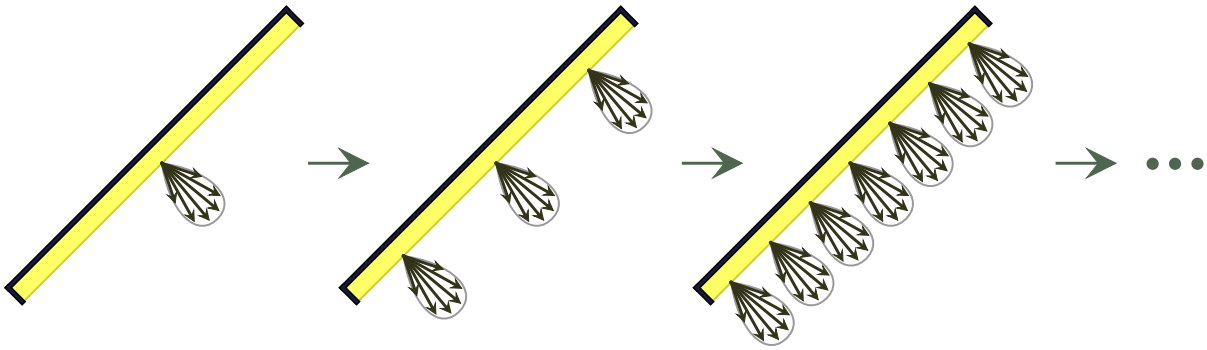


Fig. 11.4 - The total emission of the surface is the sum of the emission of the all the points on the surface

MDL also can specify what coordinate system is used as the reference framework for the evaluation of emission at points on the surface (represented by the red lines in Figure 11.5) This capability provides directional control useful in the emulation of spotlight effects.

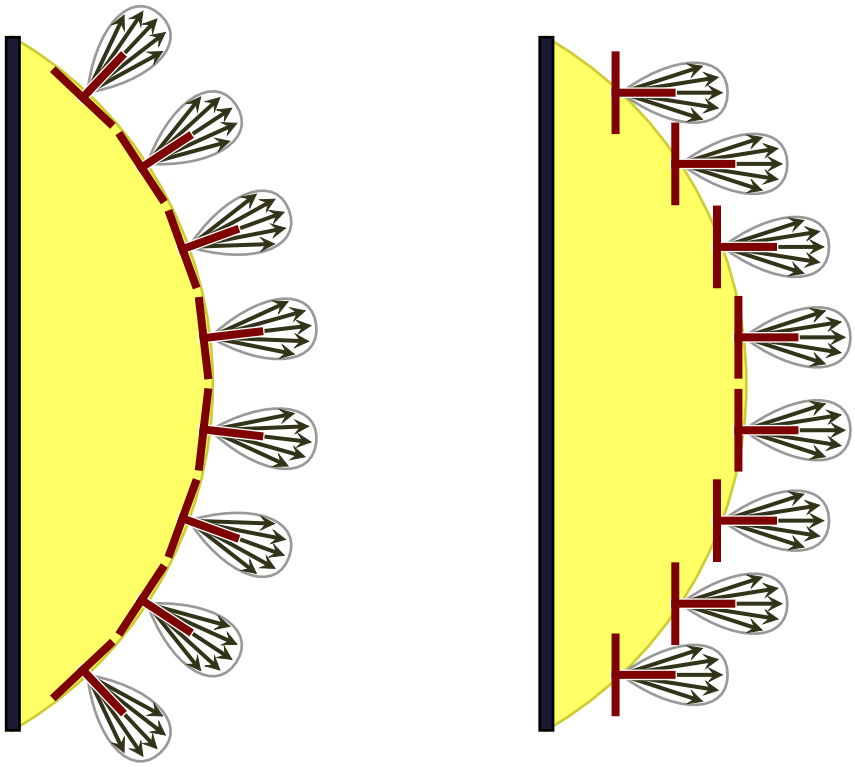


Fig. 11.5 - Light oriented by the local tangent space (left) and a global coordinate system defined for all emissive points (right).

The following sections describe the types of emission distribution functions defined by MDL and the way in which scene elements can distribute light in the Iray scene.

11.2 Emission distribution functions

The way in which light is distributed from an emissive surface is defined in the "Material Definition Language" (page 90) (MDL) by an *emission distribution function* (EDF). An EDF is specified as a field value in an MDL struct.

The `material_surface` struct defines light emission through its `emission` field:

Listing 11.1

```
struct material_surface {
    bsdf scattering = bsdf();
    material_emission emission = material_emission();
};
```

Emission field of the material_surface struct

The value of the `emission` field is an instance of the `material_emission` struct. This struct defines the EDF to use for emission in its `edf` field:

Listing 11.2

```
struct material_emission {
    edf emission = edf();   EDF field of the material_emission struct
    color intensity = color(0.0);
    intensity_mode mode = intensity_radiant_exitance;
};
```

The result of the EDF is also scaled by the `intensity` field of the `material_emission` struct, depending upon the value of the `mode` field, an `enum` of type `intensity_mode` that determines the measure used for the emission intensity:

Listing 11.3

```
enum intensity_mode {
    intensity_radiant_exitance,
    intensity_power
};
```

In general, an EDF complements the distribution function in a `material_surface` struct that defines how light is reflected from and transmitted through a surface by its `scattering` field. Taken together, reflected, transmitted, and emitted light account for all illumination perceived on an object.

11.2.1 Types of emission distribution functions

MDL defines three types of emission distribution functions:

`diffuse_edf`
: Light is emitted uniformly in all directions.

`spot_edf`
: Light emission is restricted into a cone into which the hemispherical domain for the distribution is rescaled. By default, the spot light axis is oriented along the surface normal at the emission point. A parameter defines an exponent that modifies the cosine of the angle between the axis and the sample point.

`measured_edf`
: The light distribution is defined by a *light profile*, a description of the light emittance for real-world lights from a point light source in a particular direction. A light profile is the primary parameter of the `measured_edf`. It can also be included in the scene as an argument of an MDL function call (see `IFunction_call`) or as the default argument of an MDL function definition (see `IFunction_definition`). The type of such an argument is `IType_light_profile`.

  The IES light profile [IES02] (page 197) is an industry-standard format for describing real-world lights and is implemented in MDL. These light profiles are datasets supplied by lamp vendors to describe their individual products.

11.3 Light distribution from scene elements

Iray provides the `ILight` element for the implementation of point, spot, directional and area lighting. Iray extends the notion of the environmental contribution to lighting calculation with an environment dome (page 119).

Iray also provides the ability to specify an emission distribution function in the material attached to an object instance. In this way, any object can be a source of light emission, but may also have reflective and transmissive properties as well.

The `ILight::set_type`() and `ILight::get_type`() methods of `ILight` use the `Light_type` enum to define and acquire the type of light geometry.

Field in `Light_type`	*Meaning*
`LIGHT_POINT`	Emits in all directions from origin
`LIGHT_INFINITE`	No origin, all light rays parallel

11.3.1 Point and spot lights

In Iray, point lights are positioned at the coordinate origin of the local light space. Spot lights are not defined by an enum field value of `Light_type`, but by providing a light of type `LIGHT_POINT` and using the spot_edf (page 116) distribution function. Spot lights point down the negative z-axis, like the camera. This differs from some traditional scene description formats, in which a light can have an origin, a direction, and area edge and axis vector. Instead, Iray folds this information into the light instance transformation.

11.3.2 Directional lights

A directional light is a light at infinity that emits light in the direction of the negative z-axis. A different emission direction can be set by attaching the `ILight` representing the directional light to an instance node of type `IInstance` and setting its transformation matrix.

The `LIGHT_INFINITE` type defines a direction but does not have a position, as point and spot lights do. The traditional use of a directional light is as an approximation of the rays of the sun, which is at such a great distance to the earth that at human scale the light rays are effectively parallel.

11.3.3 Area lights

Area lights specify a geometric shape that defines the source of light energy. Like directional lights, area lights are oriented through the transformation matrix of the `IInstance` to which they are attached.

The `ILight::set_area_shape`() and `ILight::get_area_shape`() methods of `ILight` use the `Light_area_shape` enum to define and acquire the shape of an area light.

Field in `Light_area_shape`	*Meaning*
`AREA_NONE`	The `ILight` instance is not an area light
`AREA_RECTANGLE`	Rectangular shape
`AREA_DISC`	Disc shape
`AREA_SPHERE`	Sphere shape
`AREA_CYLINDER`	Cylinder shape

Note: A tranformation has only limited effect on some area light shapes: In particular, a sphere shape will stay a sphere and not become an ellipse, a disc will stay a disc, and a cylinder will stay a cylinder with a circular intersection.

11.3.4 Light-emitting objects

A material defined in the "Material Definition Language" (page 90) (MDL) specifies distribution functions for light scattering and emission. In this way, any object in the scene may be defined as a light source and the light it emits is evaluated in a physically based manner in rendering. However, because the MDL material can also define scattering properties, an object can both emit and reflect light. In this manner, illumination effects like a dimly glowing glass sphere with reflections can also be implemented.

12 Environment dome

NVIDIA Iray features a flexible environment dome as an extension over the conventional infinite spherical environment. Depending on the selected mode, the dome can also be of finite size, where camera movements inside the dome provide a different environment lookup.

As an additional feature of the environment dome an implicit ground plane can be enabled, which can also act as a shadow catcher. It is also able to provide other illumination effects, such as reflections, of the objects in the scene.

12.1 Environment function

The MDL environment function is controlled with the following attribute on the `IOptions` class:

`mi::IRef environment_function`
: The environment function used for the lookup of the conventional infinite spherical environment, and also for the projections onto the different dome geometries. All MDL functions are supported whose return type is either `color` or base::texture_return. In the latter case, only the `tint` field is used and `mono` filed is ignored.

  The following MDL functions are natively supported in the render modes and allow fast updates on parameter changes:

  - `base::environment_spherical`
  - `base::sun_and_sky`
  - `base::perez_sun_and_sky`
  - `nvidia::iray::hosek_wilkie_sky::sun_and_sky`

  Other environment functions require a pre-process that bakes the function into a texture. For Iray Photoreal the baked texture drives environment light importance sampling and the MDL environment function is executed at runtime, for Iray Interactive the baked texture is used exclusively during rendering.

Simple transformations on the environment function can be applied using two further attributes on the `IOptions` class without triggering texture baking:

`mi::Float32 environment_function_intensity = 1`
: A linear scale that is applied to the intensity returned by the environment function.

`mi::Color environment_function_tint = mi::Color(1)`
: A color tint that is applied to the intensity returned by the environment function.

```
mi::Float32_3_3 environment_function_orientation =
  mi::Float32_3_3(1,0,0,0,1,0,0,0,1)
```

This matrix defines the transform from canonical directions (used for the environment function evaluation) to world space. It can be used to freely rotate the environment. The matrix needs to be orthonormal.

A number of options allow fine-tuning the baking of environment functions to a texture. As before, these options are modeled as attributes on the `IOptions` class.

`mi::Sint32 environment_lighting_resolution = 2048`
This option controls the number of pixels used for the resolution of the polar angle. The azimuthal resolution is always twice the polar resolution. Higher resolutions result in more detail in the baked environment representation. However, baking times are increased accordingly. Note that for the Iray Photoreal render mode the resolution merely determines the quality of environment light importance sampling while for the Iray interactive render mode it directly determines the quality of the environment.

`bool environment_lighting_blur = false`
Applies a small Gaussian filter kernel to blur the baked environment representation. This may increase the robustness of importance sampling in the Iray Photoreal render mode and improve the visual quality of low resolution environments in the Iray Interactive render mode.

`bool iray_use_baked_environment = false`
Forces the Iray Photoreal render mode to use the baked environment representation exclusively for rendering (similar to the Iray Interactive render mode).

12.2 Environment dome options

The dome is controlled with the following attributes on the `IOptions` class, shown here with their default settings:

`const char* environment_dome_mode = "infinite"`
The supported modes are:

`"infinite"`
Conventional infinite spherical environment lookup (default)

`"ground"`
Infinite spherical environment lookup, but with a textured ground plane,

`"sphere"`
Finite size sphere shaped dome

`"box"`
Finite size box shaped dome

`"sphere_with_ground"`
Finite size sphere shaped dome where the lower part of the environment is projected onto the plane dividing upper and lower parts of the sphere,

`"box_with_ground"`
Finite size box shaped dome where the lower part of the environment is projected onto the plane dividing upper and lower parts of the box.

For the finite size domes, the projection is computed by assuming a virtual camera position that is defined by `"environment_dome_ground_position"` offset by `"environment_dome_ground_texturescale"` times `"environment_dome_rotation_axis"`. So, as an example, if the real environment is an approximately box-shaped room and the camera position to capture the environment map is known, then those values can be used directly to set up a faithful representation of the environment. Note that for finite size domes all scene geometry should be enclosed within the dome, since shading is not guaranteed to be well-defined outside of it. When a camera ray misses a finite size dome it is shaded with the backplate background color.

`mi::Float32_3 environment_dome_position = mi::Float32_3(0,0,0)`
The origin of the dome in case it is of finite size.

`mi::Float32 environment_dome_radius = 100`
The dome radius for modes `"sphere"` and `"sphere_with_ground"`.

`mi::Float32 environment_dome_width = 200`
The dome width for modes `"box"` and `"box_with_ground"`.

`mi::Float32 environment_dome_height = 200`
The dome height for modes `"box"` and `"box_with_ground"`.

`mi::Float32 environment_dome_depth = 200`
The dome depth for modes `"box"` and `"box_with_ground"`.

`mi::Float32_3 environment_dome_rotation_axis = mi::Float32_3(0,1,0)`
This axis defines the rotation axis for interactive rotations of the dome, without the need for expensive rebaking of the environment. It also defines the up vector of the optional ground plane and can thus be used to transform the dome for z-up scenes by setting to (0,0,1). This does not affect the lookups into the environment map; the environment lookup function has to be z-up as well in this case.

`mi::Float32 environment_dome_rotation_angle = 0`
Angle of rotation for interactive rotations of the dome without the need for expensive rebaking of the environment. The range of the rotation angle is [0,1].

`bool environment_dome_visualize = false`
Enables a false-color overlay of the finite size environment geometry. This can be used to align the geometry with the projection of the captured environment map.

12.3 Implicit ground plane

An additional implicit “shadow catcher” ground plane can be specified. This plane weights the environment contribution of the ground using a progressive estimation of the amount of shadow on the ground plane, caused by the objects in the scene (from direct light only). The mode of the environment lookup itself remains unchanged as it is specified by the environment dome settings. The ground plane is a special case of the more general concept of matte objects and matte lights (page 45).

The ground plane is controlled with the following attributes on the `IOptions` class, shown here with their default settings:

`bool environment_dome_ground = false`

Controls whether the implicit ground plane is enabled or not.

`mi::Float32_3 environment_dome_ground_position = mi::Float32_3(0,0,0)`
The origin of the ground plane.

`mi::Color environment_dome_ground_reflectivity = mi::Color(-1,-1,-1,1)`
Controls the reflectivity of the ground plane at normal incidence (reflections are Fresnel-weighted). A value of zero or less deactivates ground plane reflections for the default setting of `environment_dome_ground_reflectivity_grazing`.

`mi::Color environment_dome_ground_reflectivity_grazing = mi::Color(-1,-1,-1,1)`
Controls the reflectivity of the ground plane at grazing angles (reflections are Fresnel-weighted). While a negative value is actually interpreted as white it does deactivate ground reflections in case `environment_dome_ground_reflectivity` is zero.

`mi::Float32 environment_dome_ground_glossiness = 100000.0`
Controls the glossiness of the ground reflectivity. Useful values range from 0 (diffuse) to 10000000.0 (perfect mirror).

`mi::Float32 environment_dome_ground_shadow_intensity = 1.0`
Controls the intensity of the shadow on the ground plane with a value greater or equal to 0.0. A value of 0.0 denotes no shadow at all and a value of 1.0 denotes regular shadow intensity, thus values between 0.0 and 1.0 denote a lighter shadow. Values above 1.0 increase the shadow intensity. Note that the range of useful values above 1.0 is rather limited and exceeding this range may lead to an unnatural dark shadow that can influence other additive lighting effects, such as reflections of the objects in the scene. The exact useful range is scene dependent.

`mi::Float32 environment_dome_ground_texturescale = 0`
Used to control the texture projection of the environment on the ground. This can be seen as an equivalent to the distance from ground where the real life camera was placed to take the captured environment map.

`bool environment_dome_ground_visible_from_below = true`
When set to false, one can see geometry located above the ground plane, even when the camera is placed below it.

`bool environment_dome_ground_connect_to_environment = false`
Only effective if a backplate function or the backplate color is set. When set to true, secondary interactions (like a reflection) with the ground plane will use the environment instead of the backplate/color. This is a special case of the general matte option. See "Matte objects and matte lights" (page 45).

`bool environment_dome_ground_connect_from_camera = false`
This additional flag makes the ground plane behavior toggle between two different lookup types. The first one is done locally (the flag is set to true), where similar interactions with the ground plane (for example, the same point of intersection) will deliver the exact same lookup into the real-world photograph. This usually works best for a very elaborate matte object setup specifically arranged for only a single or very similar camera position(s). The second type is global (flag is set to false), where the real interactions with the environment or backplate are used instead, which is usually preferable for simple ground plane setups (like using it only as a shadow catcher) or full

camera interaction. Note that setting the flag will only be noticable in scenes where no backplate/color at all is specified (so all interactions will always be happening with the environment), or if a backplate/color is specified and the `"environment_dome_ground_connect_to_environment"` flag is set in addition. This is a special case of the general matte option. See "Matte objects and matte lights" (page 45).

`bool environment_dome_ground_legacy_reflection = true`

When set to true, the ground will only reflect objects directly, but not when seen from reflections on other objects. Also, reflections will not replace the (optional) ground shadow, which allows to have highly glossy reflections mixed with diffuse shadowing. For a more physically correct behavior, setting the option to false will avoid this, but one has to take care to not mix strong ground shadows with a high ground glossiness (which is equivalent to the behavior of regular matte objects).

13 Camera

A camera includes physically-based properties such as the lens settings for a depth-of-field effect, but also offers convenient built-in tonemappers and the selection of a virtual backplate photograph. These are controlled with attributes of the `ICamera` class in the Iray API, described in more detail in the following sections.

The following attribute on the `ICamera` class, shown here with its default settings where applicable, controls the camera:

`bool mip_use_camera_near_plane = false`
: Controls the camera's near clip plane. When enabled the camera's hither clipping distance is used for clipping direct visibility. The near clip plane turns into a spherical clip surface when a spherical and cylindrical camera distortion is used.

13.1 Tonemapping

The different render modes share built-in tonemappers, one of them being an extension to the well known `mia_exposure_photographic` shader. These are used to map the unprocessed, simulated high dynamic range results to values that current display technology is able to work with and thus also offers optional display gamma correction. The tonemappers are disabled by default, but it is recommended to use these built-in models for best rendering performance and output quality.

The tonemapper is controlled with the following attribute on the `ICamera` class.

`const char* tm_tonemapper`
: Defines the tonemapper to use. The attribute is not defined by default and thus no tonemapper is enabled. The available built-in tonemappers are `mia_exposure_photographic` or `tm_custom_tonemapper`.

`bool tm_enable_tonemapper = true`
: Defines if the tonemapper should actually be used to tonemap the image. If disabled, it will only steer the optional firefly filtering (see `iray_firefly_filter` and `iray_nominal_luminance`)

13.1.1 Photographic tonemapper

After (optionally) applying vignetting, the `mia_exposure_photographic` tonemapper comprises three stages: first linear exposure, then compression into displayable range, and finally a set of post effects including gamma. It is controlled by the following attributes.

`bool iray_internal_tonemapper = true`
: Additional control if the internal tonemapper is enabled, assuming `tm_tonemapper` is set to `"mia_exposure_photographic"`. This is the case by default but it can be disabled if there is a clash with a tonemapper plugin which shares the parameters.

`mi::Float32 mip_vignetting = 0.0`
In a real camera, the angle with which the light hits the film impacts the exposure, causing the image to go darker around the edges. The vignetting parameter simulates this. At 0.0 it is disabled, higher values cause stronger darkening around the edges. Note that this effect is based on the cosine of the angle with which the light ray would hit the film plane, and is hence affected by the field-of-view of the camera, and will not work at all for orthographic renderings. A good default is 3.0, which is similar to what a compact camera would generate.

13.1.1.1 Exposure control

`mi::Float32 mip_cm2_factor = 1.0`
In "Photographic mode" (nonzero `film_iso`) `cm2_factor` is the conversion factor between pixel values and candela per square meter. This is discussed more in detail below.

In "Arbitrary" mode, `cm2_factor` is simply the multiplier applied to scale rendered pixel values to screen pixels. This is analogous to the gain parameter of `mia_exposure_simple`.

`mi::Color mip_whitepoint = mi::Color(1.04287, 0.983863, 1.03358, 1.0)`
Whitepoint is a color that will be mapped to "white" on output. An incoming color of this hue/saturation will be mapped to grayscale, but its intensity will remain unchanged.

`mi::Float32 mip_film_iso = 100.0`
The `mip_film_iso` should be the ISO number of the film, also known as "film speed". As mentioned above, if this is zero, the "Arbitrary" mode is enabled, and all color scaling is then strictly defined by the value of `cm2_factor`.

`mi::Float32 mip_camera_shutter = 0.125`
The camera shutter time expressed as fractional seconds, for example, a value of 100 defines a camera shutter of 1/100. This value has no effect in "Arbitrary" mode.

`mi::Float32 mip_f_number = 8.0`
The fractional aperture number. For example, the number 11 defines aperture "f/11". Aperture numbers on cameras go in a standard series: f/1.4, f/2, f2.8, f/4, f/5.6, f/8, f/11, f16, etc. Each of these are referred to as an *f-stop* or a *stop*. Each f-stop in the sequence admits half the light of the previous f-stop. Note that this shader doesn't count "stops", but actually wants the f-number for that stop. This value has no effect in "Arbitrary" mode.

13.1.1.2 Compression

The compression stage defines the actual "tone mapping" process of the image, i.e. it defines how the high dynamic range values are adapted to fit into the black-to-white range of current display devices.

`const char* mip_compression_variant = "reinhard"`
Defines the curve driving the compression to displayable range. The default value `reinhard` uses a Reinhard-style tonemapper driven by the parameter `mip_burn_highlights`, a mode which is fully compatible with the `mia_exposure_photographic` shader. Further options are `uncharted2` and `ue4_aces`,

two filmic tone mapper curve variants without parameters. Using `raw_parameters` offers most flexibility and allows to set all parameters of the internal representation.

`mi::Float32 mip_burn_highlights = 0.25`
For the compression variant `reinhard` this parameter defines how much "over exposure" is allowed. A value of 1 yields an identity mapping. As it is decreased towards 0, high intensities will be more and more "compressed" to lower intensities. When `mip_burn_highlights` is 0, the compression curve is asymptotic so that an infinite input value maps to a white output value, making over-exposure no longer possible. A good default value is 0.5.

`mi::Float32_4 mip_compression_parameters0`
`mi::Float32_3 mip_compression_parameters1`
For the compression variant `raw_parameters` these options set the parameters $p_0, \ldots, p_6$ of the tonemapping function $f(x) = (x \cdot (p_0 \cdot x + p_1) + p_2)/(x \cdot (p_3 \cdot x + p_4) + p_5) + p_6$. Note that while this offers fine-grained control of the compression characteristics, care should be taken to create meaningful compression curves to avoid rendering artifacts.

`bool mip_burn_highlights_per_component = true`
By default compression is applied separately to all channels, which can lead to saturation loss though. Disabling the parameter applies compression to the luminance, preserving the chromaticity of the output color.

`bool mip_burn_highlights_max_component = false`
Enabling this parameter applies compression depending on the maximum value of the different color components. Note that if enabled, this options overrides `mi_burn_highlights_per_component`.

`bool mip_burn_highlights_blended_component = false`
Enabling this parameter applies a linear blend between compression per color component and based on maximum color component. In this mode, the saturation of the highlights may be controlled using the option `mip_burn_highlights_saturation`. Note that if enabled, this option overrides `mip_burn_highlights_per_component` and `mip-burn_highlights_max_component`.

`mi::Float32 mip_burn_highlights_saturation = 0.0`
This parameter drives the saturation of highlights if `mip_burn_highlights_blended_component` is set.

13.1.1.3 Post effects

The effects are applied in the order listed here.

`mi::Float32 mip_saturation = 1.0`
The saturation parameter allows an artistic control over the final image saturation. 1.0 is the standard "unmodified" saturation, higher increases and lower decreases saturation.

`mi::Float32 mip_crush_blacks = 0.2`
When the upper part of the dynamic range becomes compressed it naturally loses some of it's former contrast, and one often desires to regain some "punch" in the image by using the `crush_blacks` parameter. When 0, the lower intensity range is linear, but when raised towards 1, a strong "toe" region is added to the transfer curve so that low intensities get pushed more towards black, but in a gentle fashion.

`mi::Float32 mip_gamma = 2.2`
The gamma parameter applies a display gamma correction. If the image will be displayed as-is, without further post-processing by the application, this value should be set to match the displays characteristic, otherwise disabled by setting to 1.

Note: Note that the attributes have the prefix `mip_`, but apart from that their meaning is identical to the parameters of the `mia_exposure_photographic` shader.

13.1.2 Custom tonemapper

This section describes parameters of the custom tonemapper.

`struct Tm_custom_curve tm_custom_tonemapper`
The structure of type `Tm_custom_curve` describes a mapping curve controlled by the following members:

```
mi::Float32 exposure = 0.0
Color3 whitepoint = (0.0, 0.0, 0.0)
struct Tm_custom_curve_control_point[] red
struct Tm_custom_curve_control_point[] green
struct Tm_custom_curve_control_point[] blue
mi::Float32 gamma = 0.0
```

The components of the tonemapper are applied in the order of the struct members: first exposure control and whitepoint correction are done, then a mapping of the individual color channels, and finally a gamma correction to match the display (a good default is usually 2.2).

The struct members `red`, `green`, and `blue` are dynamic arrays of structures of type `Tm_custom_curve_control_point`. This structure has the following members:

`mi::Float32 in`
The input value that is mapped to the corresponding `out` value.

`mi::Float32 out`
The output value onto which `in` is mapped.

The represented curve is a piecewise linear interpolation of the given control points. The `in` and `out` arrays need to be monotonically increasing. Note that one should disable the built-in gamma correction by setting it to 1, if the mapping curve already handles the display correction.

13.2 Bloom filter

Iray includes a built-in filter to roughly approximate a bloom/glare effect. As the filter is solely based on post-processing, it is mostly targeted at very subtle bloom and glare effects like adding slight halos to emissive objects.

The following attributes on the `IOptions` class, shown here with their default settings, control the integrated filter:

`bool iray_bloom_filtering = false`
Enable or disable the filter.

`mi::Float32 iray_bloom_filtering_radius = 0.01`
Sets the blur radius of the bloom filter, specified as fraction relative to the output resolution.

`mi::Float32 iray_bloom_filtering_threshold = 0.9`
Specifies the threshold to cutoff bright spots in the image and blur these in a second step.

`mi::Float32 iray_bloom_filtering_brightness_scale = 1.0`
Scaling factor for the blurred bright spots when composited in the third step.

13.3 Lens

The camera holds lens settings to simulate depth-of-field and other camera effects. The following attributes on the `ICamera` class control the behavior. Note that sizes etc. are represented in scene units and thus must be adjusted according to the scene dimensions.

`mi::Float32 mip_lens_radius = 0.0`
Controls the radius of the lens.

`mi::Float32 mip_lens_radial_bias = 0.5`
Control to steer the sampling of the lens and the resulting bokeh effect. For values greater than 0.5 the samples are biased towards the center of the lens, for values less than 0.5 towards the edge.

`mi::Float32 mip_lens_focus = 2.0`
Controls the focal distance to which the camera is adjusted.

`mi::Float32 mip_lens_shift_x = 0.0`
Controls the horizontal lens shift.

`mi::Float32 mip_lens_shift_y = 0.0`
Controls the vertical lens shift.

`mi::Float32 mip_lens_stereo_offset = 0.0`
Controls the distance between the center and the right camera position for stereo view. Iray can render both the right and left stereo images in one pass. When the `ICanvas` passed to the `IRender_context::render()` function has multiple layers iray will render the right and left eye in one pass and store these in respectively the first and second layer of the canvas (See "Rendering and canvases" (page 29)).

`const char* mip_lens_distortion_type = "none"`
Controls the lens distortion model. The default is a conventional perspective camera. The radial distortion models define the relationship between the undistorted radial pixel coordinate r_u and the distorted radial pixel coordinate r_d with regards to the ideal perspective camera.

The supported modes are:

`"spherical"`
Conventional spherical camera mapping.

`"cylindrical"`
Conventional cylindrical camera mapping.

`"poly3"`
Perspective camera with third order radial distortion model:
$r_d = r_u * (1 - k_1 + k_1 * r_u^2)$

`"inv_poly3"`
Perspective camera with third order inverse radial distortion model:
$r_u = r_d * (1 - k_1 + k_1 * r_d^2)$

`"poly5"`
Perspective camera with fifth order radial distortion model:
$r_d = r_u * (1 + k_1 * r_u^2 + k_2 * r_u^4)$

`"inv_poly5"`
Perspective camera with fifth order inverse radial distortion model:
$r_u = r_d * (1 + k_1 * r_d^2 + k_2 * r_d^4)$

`"ptlens"`
Perspective camera with fourth order radial distortion model:
$r_d = r_u * (1 - k_1 - k_2 - k_3 + k_1 * r_u + k_2 * r_u^2 + k_3 * r_u^3)$

`"inv_ptlens"`
Perspective camera with fourth order inverse radial distortion model:
$r_u = r_d * (1 - k_1 - k_2 - k_3 + k_1 * r_d + k_2 * r_d^2 + k_3 * r_d^3)$

`"brown"`
Perspective camera with fifth order radial distortion model:
$r_d = r_u * (k_1 + k_2 * r_u + k_3 * r_u^2 + k_4 * r_u^3 + k_5 * r_u^4)$

`"inv_brown"`
Perspective camera with fifth order inverse radial distortion model:
$r_u = r_d * (k_1 + k_2 * r_d + k_3 * r_d^2 + k_4 * r_d^3 + k_5 * r_d^4)$

`"equidistant"`
Simulates an equidistant fisheye with a fourth order radial distortion model:
$r_u = k_1 + k_2 * r_d + k_3 * r_d^2 + k_4 * r_d^3 + k_5 * r_d^4$

The distortion formula is used to map the distance from the image center to the exit angle. Thus, the polynomial determines the entire field of view instead of merely encoding the deviation from the ideal model. Consequently, the camera's focal length and sensor width (`ICamera::get_aperture`()) are ignored.

In order to obtain an ideal equidistant fisheye, all distortion parameters should be set to 0, except for k_2, which determines the field of view. Assuming a desired field of view of θ, sensor aspect ratio a, and `mip_lens_distortion_scale` s, k_2 should be set as follows:

Lens type	*Field of view* θ	k_2
circular fisheye	vertical	$a\theta/s$
cropped	horizontal	θ/s
full-frame fisheye	diagonal	$\frac{1}{\sqrt{\frac{1}{a^2}+1}}\theta/s$

The horizontal field of view for a physical fisheye lens which uses equidistant projection is determined by $\theta = w/f$, where w denotes the sensor width (`ICamera::get_aperture`()) and f the focal length. Vertical and diagonal field of view are computed accordingly.[1]

By default, all pixels of the image will be rendered. In order to simulate the limited image circle of physical lenses, which yields the typical black envelope for circular and cropped fisheye lenses, `mip_lens_max_fov` should be set to θ.

`mi::Float32_3 mip_lens_distortion_params = mi::Float32_3(0,0,0)`
Controls the parameters k_1, k_2 and k_3 in the radial distortion models.

`mi::Float32_3 mip_lens_distortion_params2 = mi::Float32_3(0,0,0)`
Controls the parameters k_4 and k_5 in the radial distortion models. The third component is ignored.

`mi::Float32 mip_lens_distortion_scale = 1.0`
Controls the relative scale of the lens distortion.

`mi::Float32 mip_lens_max_fov = 2.0 * M_PI`
Controls cutoff angle for rays in the `"equidistant"` fisheye camera model. Rays beyond this field of view will not be rendered and yield black pixels.

This setting is used to achieve the common black envelope around images taken with circular fisheye lenses.

An aperture function is used to lookup the lens aperture tint. The aperture function is evaluated over a square lens with the radius specified by the mip_lens_radius option. The 2D lens coordinates are passed to the function as the first texture space in the rendering state, ranging from 0 to 1. All MDL functions are supported whose return type is either `color` or base::texture_return. In the latter case, only the `tint` field is used and `mono` filed is ignored.

The aperture function is controlled with the following methods on the `ICamera` class:

```
mi::Sint32 ICamera::set_aperture_function(const char* name)
const char* ICamera::get_aperture_function()
```
Defines the texture through a texture lookup function represented as a DB element of type `IFunction_call`. There is no aperture shader by default, in which case the aperture is not tinted.

Alternatively, if no aperture function is set a simple approximation of typical camera aperture blades can be used. This is controlled by two options:

`mi::Sint32 mip_aperture_num_blades = 0`
For values greater than two this activates a blade-shaped aperture with the specified number of blades.

`mi::Float32 mip_aperture_blade_rotation_angle = 0.0`
Rotation of the blade segments in radians.

1. Note that this implies the following relationship, which explains why the sensor size does not affect depth of field in this camera model. Keeping the polynomial (and thus the field of view) constant and scaling the sensor width implies that the (implicit) focal length is scaled by the same factor. Absent any change to the lens radius, this also implies that the f-stop is scaled accordingly.

13.4 Virtual backplate

A virtual backplate is specified by a texture and a background color. All primary camera rays that would usually end up in the environment dome, perform a lookup in the backplate instead.

All lighting effects (illumination, reflection, refraction, etc.) are in general not influenced by the backplate. There is one exception though: perfectly specular, thin walled transmitting materials can be used to make the backplate appear through windows. For example if both the windshield and the back window of a car are modeled as thin walled glass, looking through both yields a backplate lookup instead of the environment dome.

It is also possible to enable this effect for perfectly specular, non-thin walled materials. This is achieved by setting the following attribute on the `IOptions` class.

`bool refract_backplate = false`
: Forces perfectly specular, non-thin walled materials to respect the virtual backplate. Note that this can lead to visible discontinuities on complex geometric shapes due to the mixture of both the backplate (texture and color) and environment dome contributions.

The backplate texture is scaled to exactly fit the rendering resolution so that rendering an empty scene will display the texture at full scale. By setting its *uv* transform, the scale and offset of the texture can be modified. In this case, the wrap mode parameters control the tiling of the texture. In regions where the transformed texture does not cover the viewport, including repetition controlled by the wrap mode, either the backplate background color is displayed or the environment lookup is used if the wrap mode and clip parameters are set to values that clip the texture.

By default, the backplate is not tonemapped, so that the backplate image in the final tonemapped rendering looks exactly like the backplate provided as input. Optionally, the tonemapper used on the rendered image, see Section can also be enabled for the backplate. These tonemappers can include a gamma correction, which is not seen as part of the tonemapping itself, and is always applied to the backplate.

> **Note:** The regular texture pipeline has support for inverse gamma correction, which can compensate the gamma correction. If needed, the setting for the inverse gamma correction can be explicitly controlled with the `ITexture::set_gamma`() and related methods.

The virtual backplate is controlled with the following methods on the `ICamera` class.

`mi::Sint32 ICamera::set_backplate_function(const char* name)`
`const char* ICamera::get_backplate_function()`
: Defines the texture through a texture lookup function represented as a DB element of type `IFunction_call`. State access in the texture function is limited to state::texture_coordinate(0). There is no backplate by default, in which case the environment dome lookup is used.

`void set_backplate_background_color(const Color_struct &color)`
`Color_struct get_backplate_background_color()`
: Defines the background color if no backplate texture is specified, or fills the regions of the viewport that a transformed texture would not cover if the parameters of a base::file_texture backplate function indicate that the texture is clipped. The default is

`mi::Color(-1,-1,-1,-1)` which disables the usage of the background color. Note that the alpha value is ignored and filled with 0 instead.

```
void set_backplate_tonemapping_enabled(bool flag)
bool get_backplate_tonemapping_enabled()
```

Tonemapping is disabled by default on the backplate. It can be controlled using these methods. The gamma correction in the tonemapper is not controlled through this flag and is always applied to the backplate as well.

```
void set_backplate_lens_effects_enabled(bool flag)
bool get_backplate_lens_effects_enabled()
```

Lens effects such as depth of field and distortion are disabled by default on the backplate. It can be controlled using these methods. Note that lens effects can not be disabled for backplate meshes.

The backplate can also be mapped onto a mesh with the following attributes on the `IAttribute_set` class (for objects and instances in the scene graph), so that one get parallax effects while navigating inside of a virtual 3D backplate.

```
bool backplate_mesh = false
```

If set to true the respective object or instance is marked as special backplate mesh. Note that the mesh must have valid texture coordinates as otherwise the backplate image cannot be mapped onto it. Also note that texture transformations are ignored for the backplate mesh; backplate mesh texturing is entirely controlled by the mesh's texture coordinates.

```
mi::IRef backplate_mesh_function
```

If set, the specified texture lookup function (of type `IFunction_call`) will be used instead of the backplate function specified for the camera. This enables more flexible texturing workflows in case of multiple backplate meshes.

13.5 Motion blur

Motion blur simulates a shutter effect of real cameras, and results in the blurred appearance of moving objects and light sources. Also movement of the camera itself causes a blur of the rendered scene. It is enabled if the shutter close time is strictly larger than the shutter open time and if the `progressive_samples_per_motion_sample` attribute is set to a value larger than one. The actual motion is set with motion transformations, see the SRT mode on the `IInstance` class. The perceived blur thus depends on the shutter time of the camera and the amount of motion that happens within this timeframe.

Motion blur is controlled with the following methods and attributes on the `IOptions` class, shown here with their default settings:

```
void IOptions::set_shutter_open(mi::Float32 shutter_open)
void IOptions::set_shutter_close(mi::Float32 shutter_close)
```

Methods on the `IOptions` class to control the shutter times. To enable motion blur, the shutter close time must be strictly larger than the shutter open time. The normal range is (0, 1), which uses the full length of the motion vectors or motion vector paths. It can be useful to set both times to 0.5, which disables motion blur rendering but computes scene data at an offset of one half of the frame, which allows bi-directional post-motion-blur in the result image. The default is 0.0 for both values, disabling motion blur.

`mi::Sint32 progressive_samples_per_motion_sample = 8`

Number of sub-frames rendered before the scene is updated to a different time value. A higher value optimizes render performance at the cost of a higher number of total frames required before the separate motion step samples become invisible in the image. A lower value increases interactivity, but reduces total render performance (for example, rendering more frames will take longer). A high value makes noise vanish faster, but motion converge slower — and vice versa. This relationship can be tweaked in one or the other direction, depending on the lighting complexity and the amount of motion specified for a scene. Only powers of 2 are valid. A value of 0 switches off motion blur.

14 Physically plausible scene setup

A primary goal of Iray is to provide artists, designers, architects, and other design professionals with the ability to easily create photorealistic imagery. No sophisticated knowledge of the theory behind 3D rendering is required; Iray, and in particular its "Iray Photoreal" (page 43) render mode, provides excellent results right out of the box.

However, as a physically-based renderer, Iray requires scenes to be set up in a physically plausible manner. To set up a scene that will yield great photorealistic results with Iray, it is sufficient to honor a few recommendations, which are provided in the following sections.

Furthermore, preparing physically plausible scenes also introduces performance benefits: potential sources of noise and other unexpected situations are implicitly avoided when geometries and materials are set up optimally.

14.1 Recommendations for geometry preparation

In the next sections you are going to get recommendations for your preparation of geometry for rendering with Iray. Avoid common pitfalls such as glass objects with no thickness and under-tessellating your geometry. Instead, stay faithful to reality: Use glass with real thickness where appropriate, use fine tessellation and pay attention to the layout of your UV coordinates for anisotropic reflections.

14.1.1 Modeling glass

In real life, glass objects always have a thickness. Even when really thin, their depth is often not inconsiderable. In order to compute proper refractions, Iray expects light rays to travel through solid geometry.

In some cases, the thickness really is negligible. Examples are far-away windows or soap bubbles, cases where refraction effects are no longer visible. In such cases, modeling objects without a thickness is fine, if the materials are set up accordingly. The thin-walled feature of MDL is designed for just this purpose. Remember that refractions will be ignored with this setting enabled.

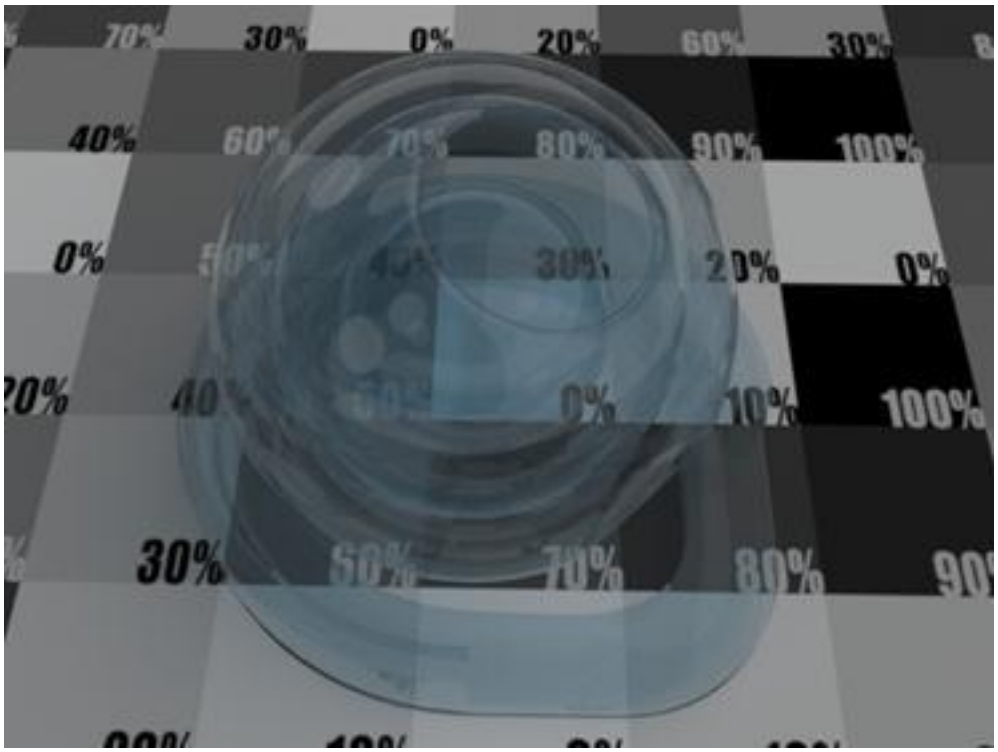


Fig. 14.1 - Thin-walled glass material

Iray Photoreal will be able to properly compute and render refractions when glass is modeled as a solid geometry.

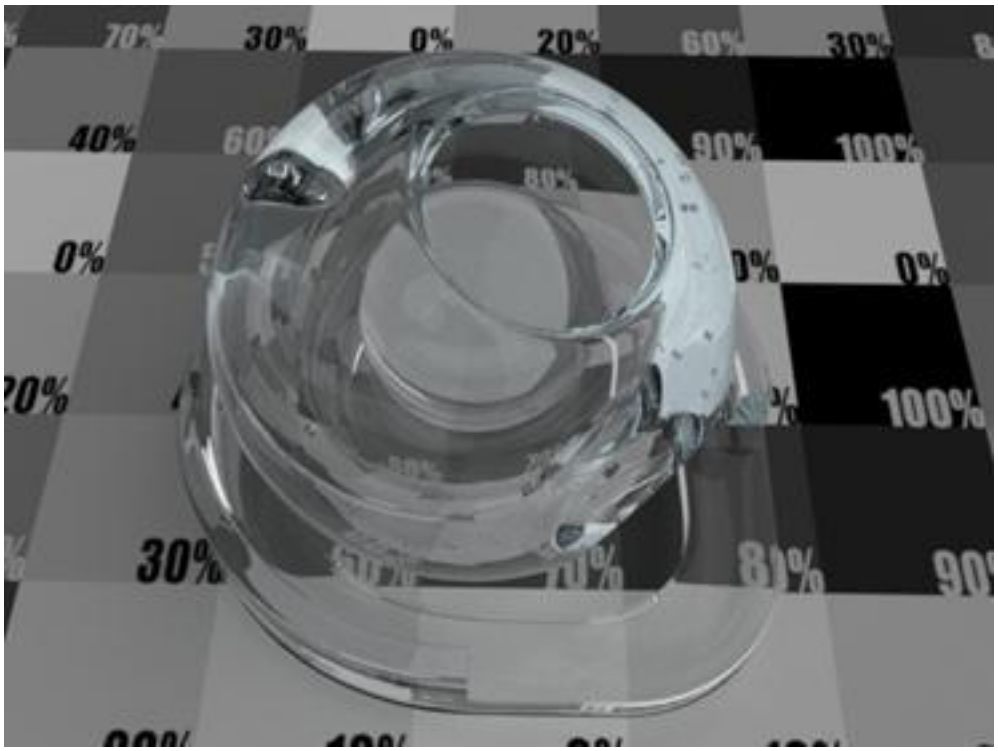


Fig. 14.2 - Glass material on a solid geometry

The following example illustrates how important it is to model glass geometries in the exact same way as they appear in real life in order to get an accurate result with Iray Photoreal.

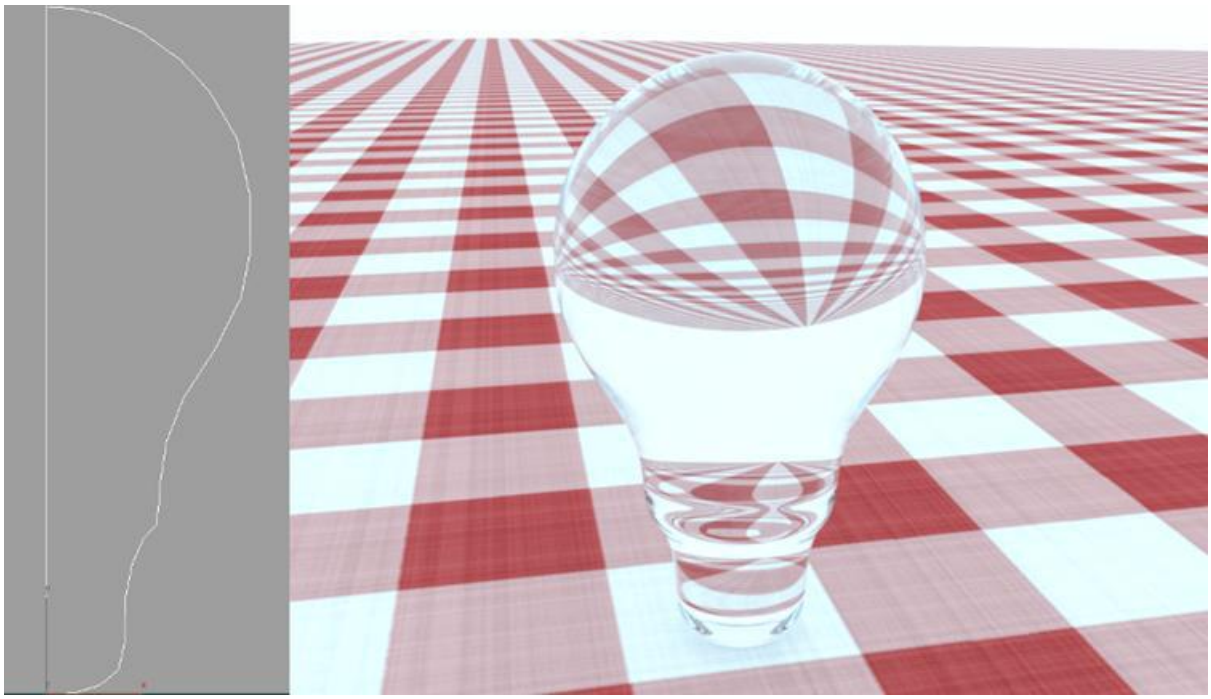


Fig. 14.3 - Light bulb modeled with a single continuous line.

In Figure 14.3, once the Lathe modifier is applied, the result will be a solid geometry. When rendered with a physical glass material, the bulb will look like its one solid piece of glass.

To model a realistic bulb, ensure that the hull is one thin layer of glass that always has a thickness. Model a very thin external layer.

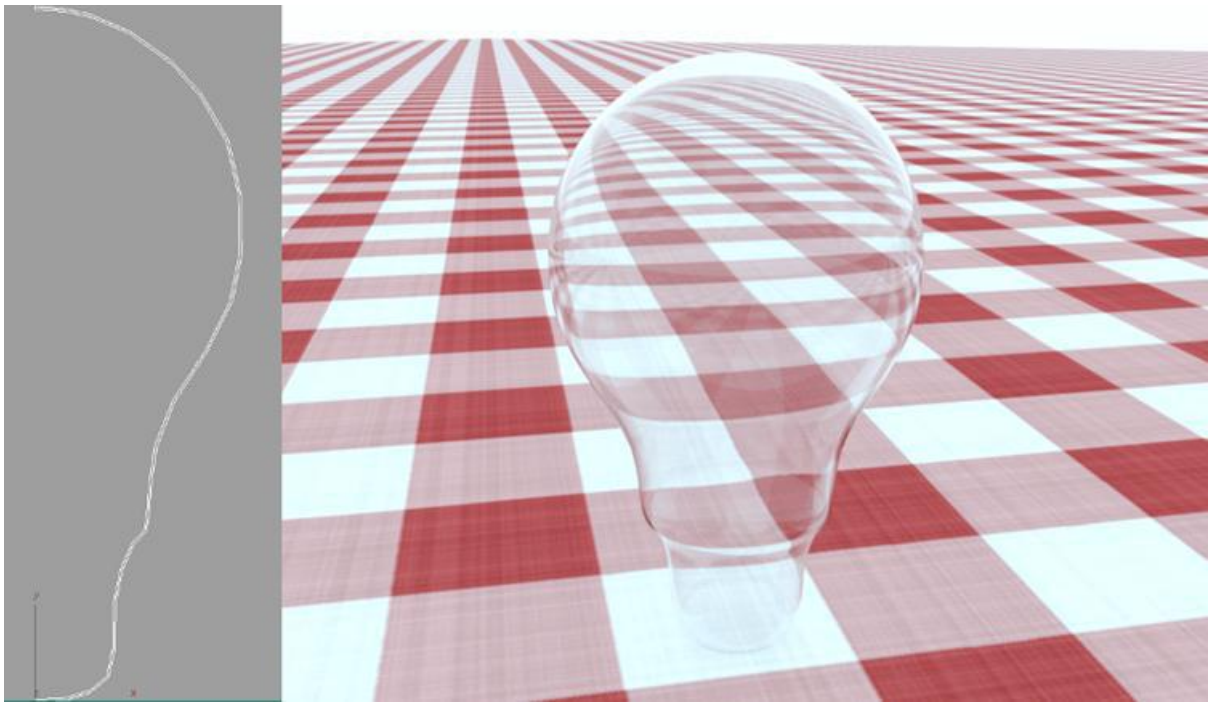


Fig. 14.4 - Physically correct modeled lightbulb.

Note in Figure 14.4 the drastic difference between the solid lightbulb above and this version which uses a modeled hull.

When rendered with a physical glass material, the result will now be photorealistic. Note the strong refraction effects at the top of the bulb.

14.1.2 Modeling volumes

This section provides recommendations about modeling neighboring volumes and enclosed volumes:

- Neighboring volumes (page 137)
- Enclosed volumes (page 138)

> **Note:** When modeling volumes, be sure that they are solid and closed.

14.1.2.1 Neighboring volumes

When modeling neighboring volumes such as liquids in basins or different layers of liquids, you can:

- Separate them with a thin layer of air, which will cause additional refraction effects as shown in the following figure:

Fig. 14.5 - Non-overlapping volumes

In Figure 14.5, the layer of air that separates them causes non-realistic refraction effects

- Slightly overlap them to avoid additional refraction effects as shown in the following figure:

Fig. 14.6 - Overlapping volumes, which shows the physically correct variant

A slight overlap is the recommended modeling technique for neighboring volumes.

14.1.2.2 Enclosed volumes

When modeling enclosed volumes, no special modeling guidelines need to be considered to ensure correct rendering results. The following line drawing provides a simple example — a glass of water with ice cubes that contain air bubbles:

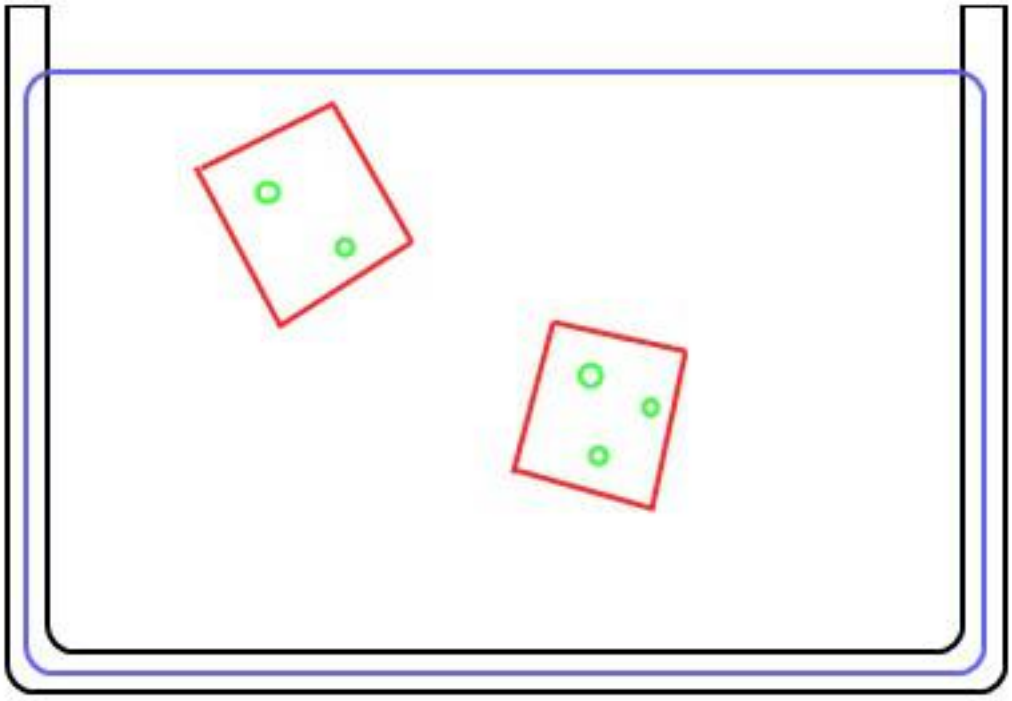


Fig. 14.7 - Ice cubes in the water render correctly with non-overlapping volumes

When modeling the air bubbles, the ice cubes and the water, it is not necessary that these overlap as the air bubbles are enclosed by the ice cubes, which are again enclosed by the water. The glass and the water are neighboring volumes however, so they should be modeled with a slight overlap.

14.1.2.3 Defining priorities

While by default an enclosed volume takes precedence over the enclosing volume, for the region of overlap this is actually not strictly defined. The underlying assumption is that the overlap is reasonably small and Iray simply uses one of the two surfaces for media separation and ignores the other.

In case the overlap is larger or in general more complex (e.g. a drinking glass with fine geometric details), priorities can be specified to explicitly define the order. For example, an ice cube floating on water should be assigned a higher priority than the water (such that it realistically displaces the water below it). Similarly, the glass containing both the cubes and water may be assigned a higher priority than the ice.

Volume stack priority is specified using an attribute on `IAttribute_set`:

`mi::Sint8 volume_priority = 0`
: Determines which objects take precedence over others in the volume stack. The scene-surrounding empty space implicitly uses the minimum priority -128.

14.1.3 Mapping uv-coordinates

UVW-coordinates are used to define how a texture is mapped on to some geometry. However, UVW-coordinates are also used when you have a surface with anisotropic

reflections. The layout of the UVW-coordinates defines the direction of the reflections on the surface. Just imagine it as the brush direction in case of brushed metal. Ensure that the UVW-coordinates are well placed for anisotropic reflections. The orientation of the mapping effects the orientation of the anisotropic reflection.

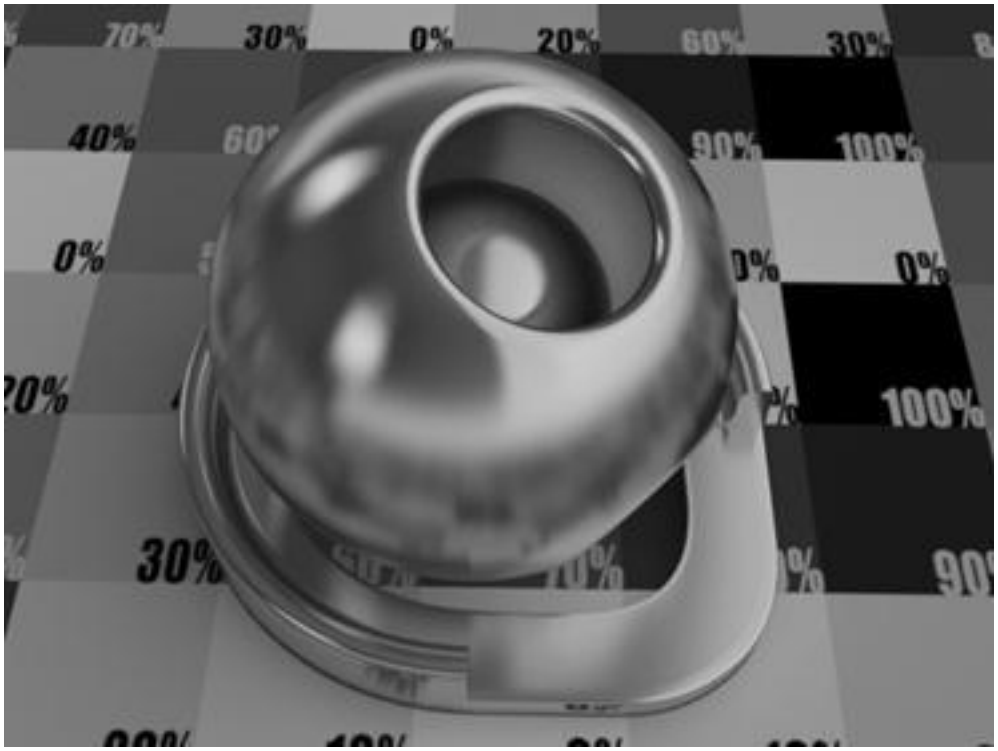


Fig. 14.8 - Anisotropic material on an object with planar UVW-mapping.

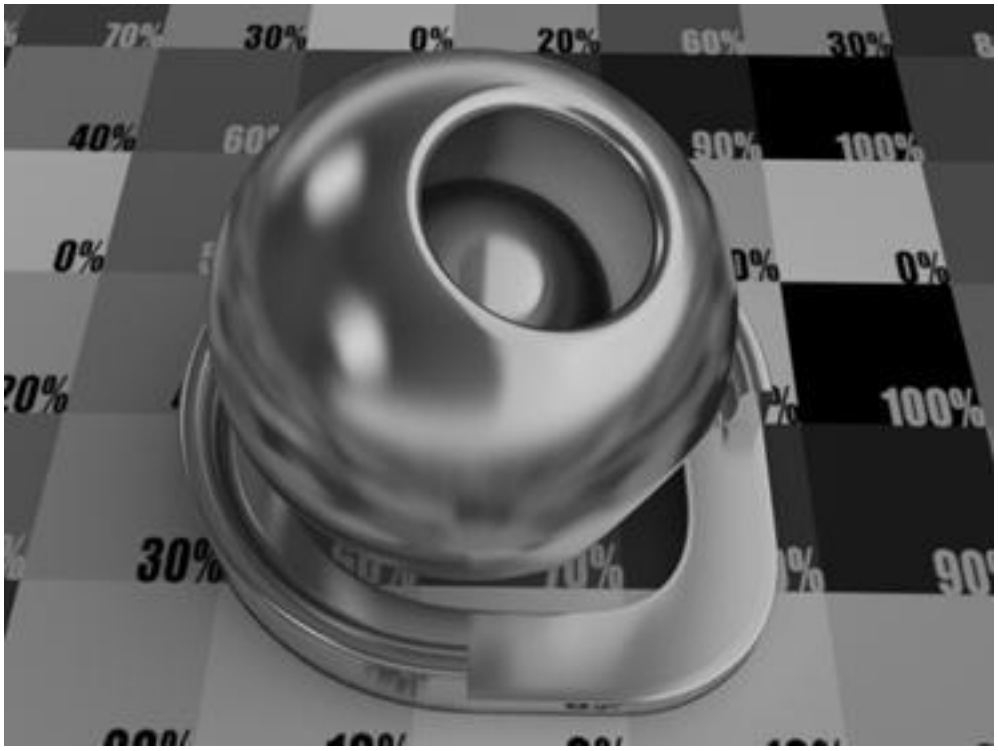


Fig. 14.9 - Anisotropic material on an object with spherical UVW-mapping.

14.1.4 Tessellating curved surfaces

To appear smooth when rendered, smooth curved surfaces must be tessellated with enough detail. Especially in areas of high curvature shading artifacts may occur if the geometry is not sufficiently tessellated. Shadow edges appear to be blocky; faces are shaded when they shouldn't be. This phenomenon is called *shadow terminator artifact*. If the silhouette looks under-tessellated, it is likely that shadow terminator problems will occur as well. Increase the tessellation of the geometry to avoid shadow terminator artifacts.

Optionally, the per-object attribute on the `IAttribute_set` class can be set to avoid most of such visible artifacts:

`bool shadow_terminator_offset = false`
: If set to true, the automatic handling of most common shadow terminator artifacts is enabled. Note that the effect is very limited if the "Caustic sampler" (page 54) is also enabled.

Note though that this automatic handling can create new kinds of problems: One example are dark spots if geometry is close-by (like in concave corners) or artifacts on refractive objects. Thus a sufficient tessellation is still recommended in all cases.

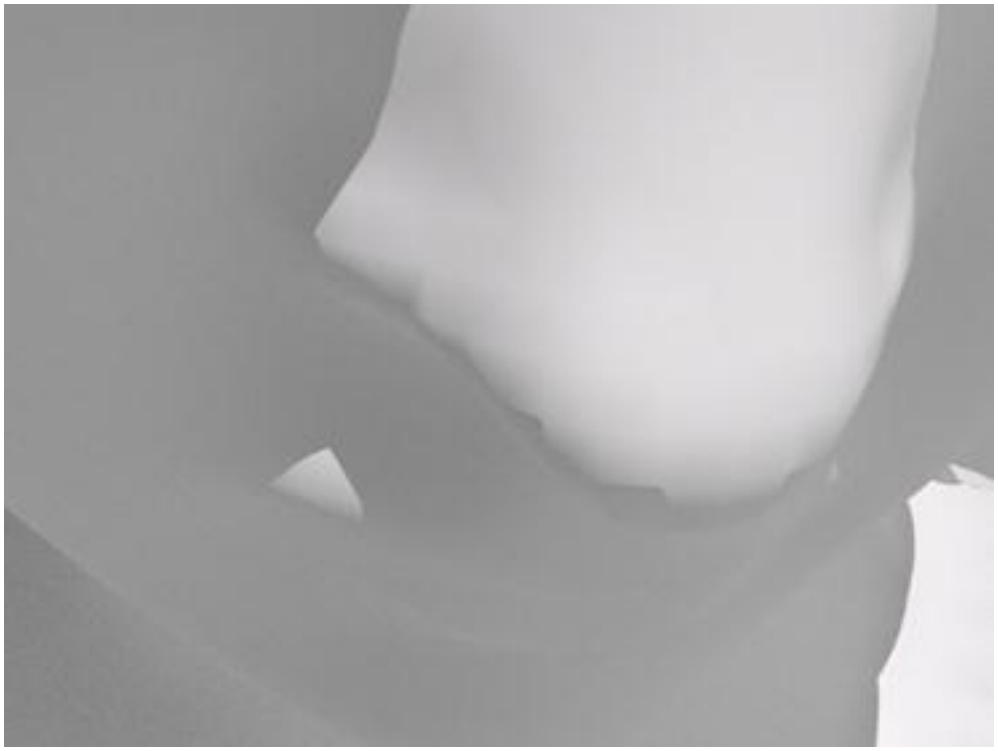


Fig. 14.10 - Poorly tessellated geometry showing bad shadow artifacts at the edge of the nose.

Fig. 14.11 - Same object with finer tessellation.

Fig. 14.12 - Poorly tessellated objects do not look good in general.

Fig. 14.13 - Same object with proper tessellation.

14.2 Recommendations for lights

Iray simulates light transport following the rules from physics. Therefore in Iray, all light sources contribute to the direct and indirect illumination of all objects. It is not possible to constrain the contribution of lights to specific objects because it distorts real-world behavior. The next chapters give you some tips for working with photometric lights.

14.2.1 Using photometric lights

Because Iray uses physically-based algorithms, if you work with realistic photometric input values in SI units, like luminous intensity values in candela, you will get realistic output results in SI units. Using physical units makes tweaking light parameters also more intuitive.

Real-world lights don't shine with the same brightness in all directions but brightness changes in different directions. Such distributions are stored in light profile files. One example are IES light profiles that store a 3d representation of the light intensity distribution. You can load those light profiles within a photometric light in Iray to render with light distributions from real-world lights. To obtain IES light profiles, visit the web page of the light manufacturer. Most professional light manufacturers provide downloadable IES files for their lights.

Lights using measured light profiles are a good way of achieving convincing lighting effects. Note, however, that such measurements assume a point light shape, which tends to yield unnaturally hard shadows. This can be ameliorated by using small area light shapes like disc and rectangle. Note the `global_distribution` parameter when using area light shapes and that the shape can actually support the emission characteristics, for example, a rectangle cannot emit light in tangent directions.

Iray supports the following types of Emission Distribution Functions (EDFs):

- Light profile (df::measured_edf)
- Spotlight (df::spot_edf)
- Uniform diffuse (df::diffuse_edf)

and light shapes:

- Point
- Rectangle
- Disc
- Sphere
- Cylinder

In addition, any geometry in the scene can be turned into a light source using the emission property of MDL materials.

14.2.2 Using environment and sunlight

Iray considers the environment for illumination contribution. Good HDR environments are likely to contribute lighting information that results in very realistic final results.

When using environment lighting, it is important to consider these recommendations:

- When using sunlight, you can use the MDL function base::sun_and_sky as environment function. The correct settings to work with luminance values that are mostly consistent with those of real world sun and sky light on a clear day are:

```
rgb_unit_conversion: color(1.0, 1.0, 1.0)
sun_disk_intensity: 1.0
multiplier: 0.10132
physically_scaled_sun: true  // default value
```

- Avoid HDR images where the sun is not as bright as in real life. Otherwise the resulting image looks washed out. To determine whether the sun is bright enough, open the HDR with an image tool and take a look at the value of the sun. A sun of realistic size should be a few thousand times brighter than other parts of the image. If its intensity value is anywhere below 1000 then the sun is too dark, and the resulting light simulation will look dull and lack the necessary contrast in the rendered image.
- Avoid low resolution HDRs, as the aliasing of small bright spots in the environment map will appear in the final picture and yield artifacts, especially in the shading/lighting of objects. Apply the optional blur settings to low resolution HDR images.

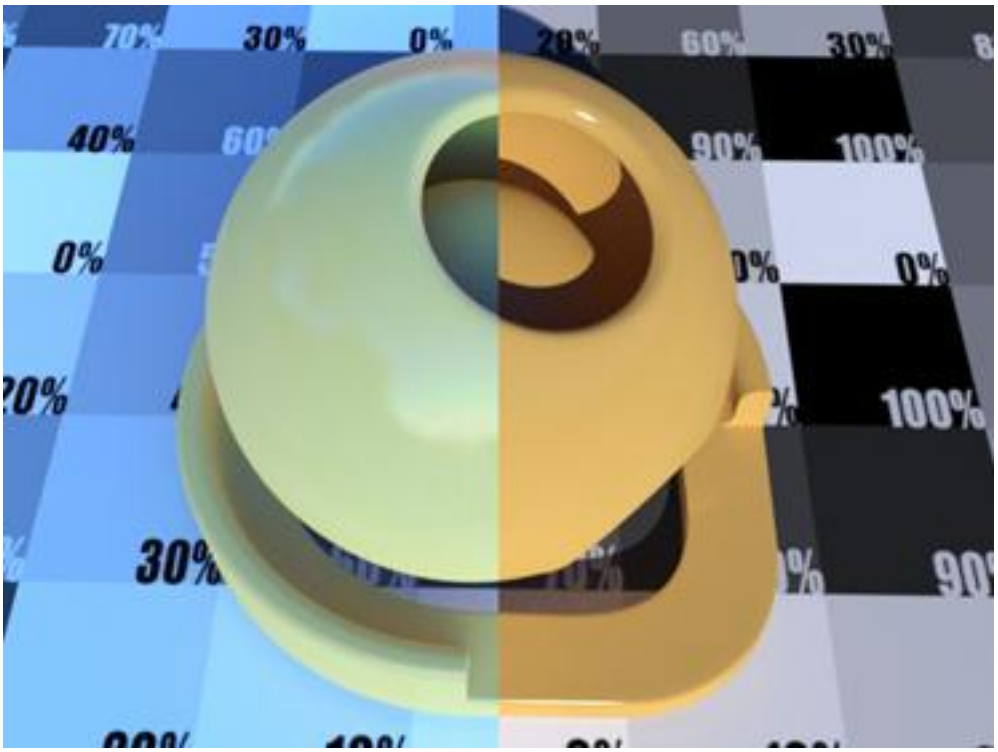


Fig. 14.14 - HDR with dull sun compared with compared with HDR with bright sun

14.3 Recommendations for materials

The following tips can help you to provide physically plausible material settings for Iray and benefit from photorealistic results with less noise and faster rendering times.

- Don’t use pure white colors. Pure white materials are rarely found in nature. They should be avoided. Avoid diffuse color values such as RGB=[1,1,1] (pure white), which could, for example, be replaced with [0.7, 0.7, 0.7] for white paper. Only few materials, such as pure snow, can reach values as high as 0.9.

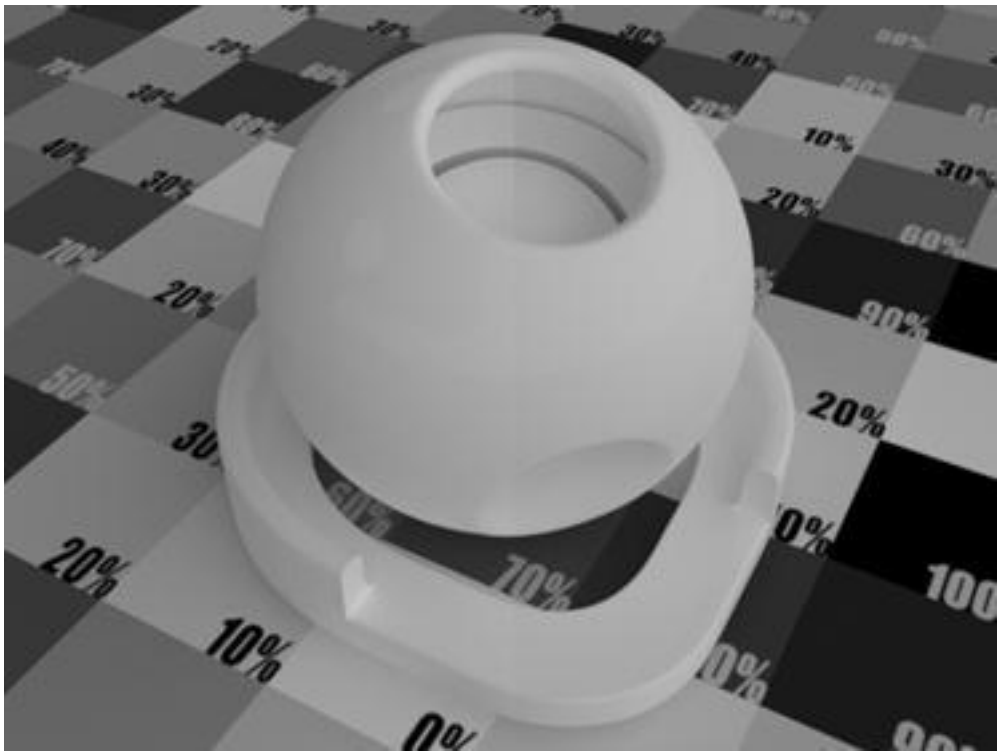


Fig. 14.15 - Pure white (left) compared with physically realistic white (right)

- Don't use pure black colors: Purely black surfaces are rarely found in nature. They should be avoided as such. An RGB value like [0,0,0] could for example be replaced with the value [0.04, 0.04, 0.04] for charcoal or fresh asphalt.

Fig. 14.16 - Pure black (left) compared with physically realistic black (right)

- Avoid perfectly reflective materials when possible. In nature, a reflectivity ratio of 0.7 is already considered high. Set reflectivity to [0.7, 0.7, 0.7] instead of pure white. Freshly

polished silver or chrome can reach reflectivities above 0.9, but oxidation and dust reduces this already.

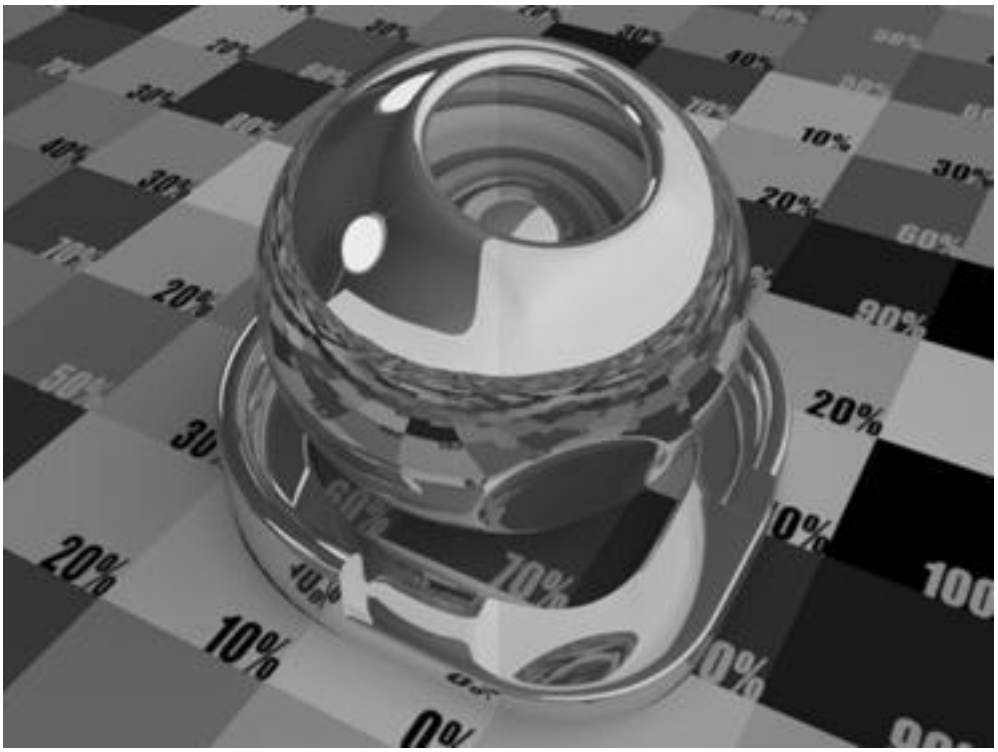


Fig. 14.17 - Perfect reflectivity (left) compared with realistic reflectivity (right)

- Decide if emitting objects have to also reflect light. In most cases, this will not be necessary and materials without any BSDFs are cheaper to compute. If you do need a full material, test its setting first before adding emission. Otherwise, the brightness of the emission will make it hard to spot problems with the material setup.

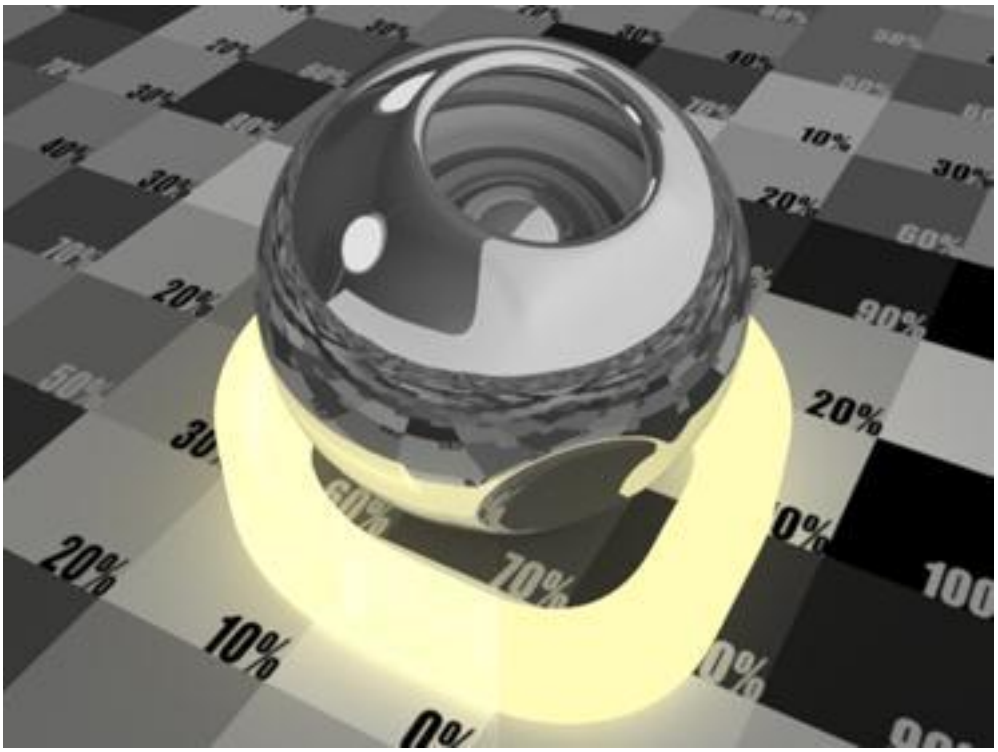


Fig. 14.18 - Improperly set up glowing material (left) compared to properly set up glowing material (right)

- Alpha maps embedded in textures are not used to automatically create holes. Use the cutout opacity feature of the MDL materials to create holes in objects.

Fig. 14.19 - Cutout opacity feature

14.4 Recommendations for cameras

14.4.1 Depth of field

To increase photorealism, you can set the camera's depth of field (page 124). Using depth of field does not add to the computational cost of rendering a scene.

Figure 14.20 is a scene rendered without depth of field calculations.

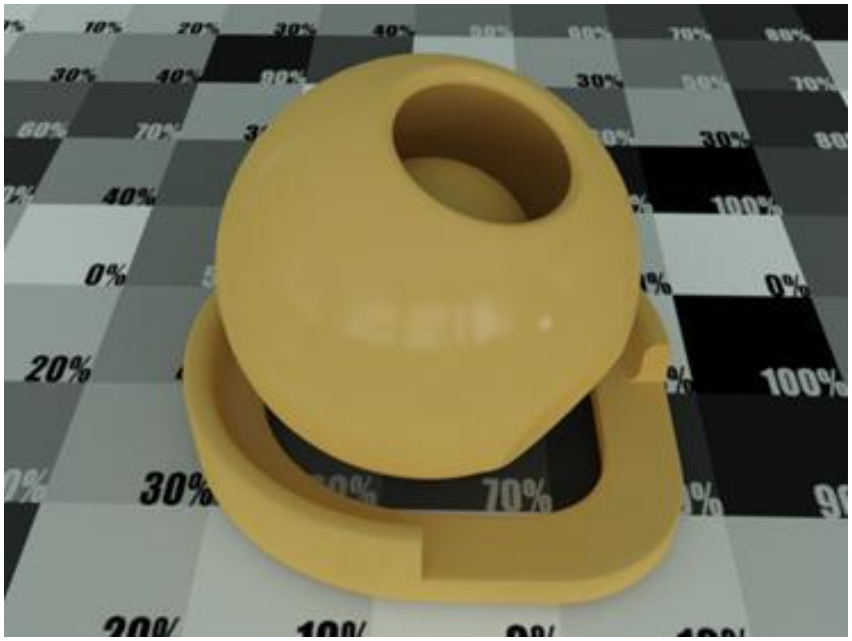


Fig. 14.20 - Without depth of field

Figure 14.21 shows the addition of depth of field.

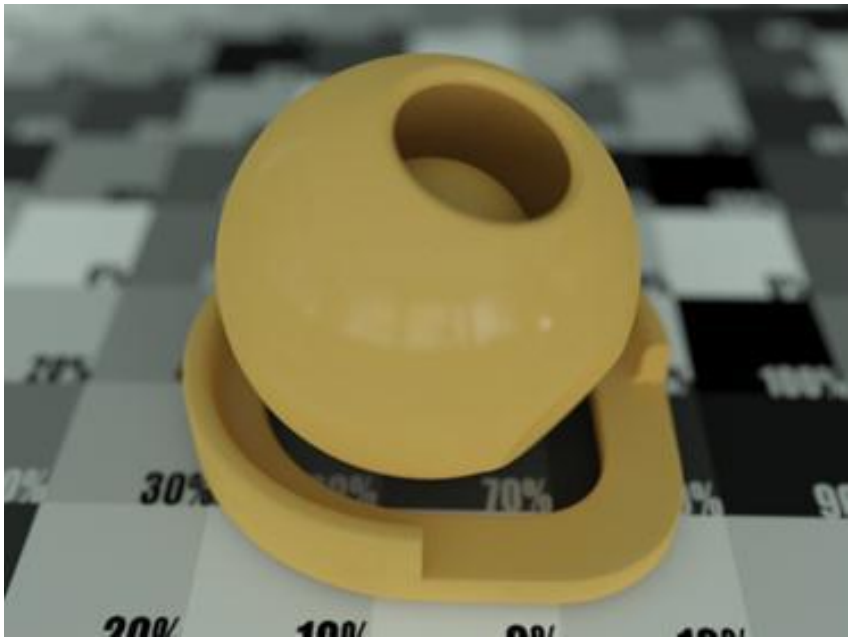


Fig. 14.21 - With depth of field

15 Networking

The following sections provide an overview about the networking support provided by Iray, supported networking modes, and Iray Bridge which allows simple and efficient use of resources of a remote cluster.

15.1 Networking overview

neuray is a distributed system supporting different networking modes, for example, UDP multicast and TCP unicast with or without automatic host discovery. Such a cluster of connected hosts uses a distributed database with redundancy to distribute data and jobs, for example, scene data and rendering requests. The configuration of the networking facilities is flexible and left to the user to support different use cases, for example, a master/slave setup, or a setup of equal peers sharing resources.

neuray provides a built-in HTTP server framework that can be used to implement a HTTP server that executes C++ code in reaction to HTTP requests (including serving of static files). This HTTP server can for example be used by an integration to provide remote viewing capabilities. neuray also provides a built-in RTMP server framework to provide RTMP streams of rendered images.

For network rendering in local clusters it is required that all files are available on all involved hosts. The absolute filenames do not necessarily need to be identical on all hosts, but the relative filenames based on the configured search paths need to be identical on all hosts.

15.2 Networking modes

This section introduces:

1. The different networking modes:
   - No networking (page 151)
   - UDP mode (page 151)
   - TCP mode (page 151)
   - TCP mode with discovery (page 151)
2. The program `example_networking.cpp` (page 244), which shows the basic configuration options to set up the different networking modes. Further configuration options related to networking are provided by the `INetwork_configuration` interface. A simple implementation of the `IHost_callback` interface is used to track network changes.

See "Networking configuration" (page 26) for a general discussion about the advantages and disadvantages of the available networking modes.

15.2.1 No networking

This mode is the default, since it does not require any additional configuration. In this mode each instance of the Iray API is separate. There are no attempts to connect to other instances. This mode was already used in the previous examples. It is part of this example program just for a matter of completeness.

15.2.2 UDP mode

The UDP mode is the simplest networking mode (apart from no networking at all). All it requires is a UDP multicast address and port number. All hosts with instances of the Iray API than can be reached via this multicast address join a single cluster. The cluster can be restricted to certain sets of hosts by using an optional discovery identifier. This option allows a fine-grained control over potential members of a particular cluster.

15.2.3 TCP mode

In TCP mode it is necessary to specify all hosts participating in the cluster in advance. Apart from your address and port, you have to specify the addresses and ports of all other hosts. Since this mode is not very flexible it is recommended to use TCP mode with discovery instead.

15.2.4 TCP mode with discovery

This mode is similar to TCP, but there is a discovery mechanism which eliminates the need to specify the addresses and ports of all other hosts. There are two variants of this mode: discovery via multicast or discovery via a master node. In the multicast variant you have to specify a common multicast address which is used by all hosts during the discovery phase to join the cluster. In the master variant you have to specify a common unicast address. The host with the given IP address acts as master during the discovery phase, all other hosts connect to this host to join the cluster. In both variants TCP is used as in the TCP mode without discovery after the host connected to the cluster. Similar to the UDP mode an optional discovery identifier can be used to restrict the cluster to a certain set of hosts.

15.2.5 Example – Starting the API in networking mode

Source code
`example_networking.cpp` (page 244)

This example demonstrates the usage of various networking modes by specifying the values of its command-line arguments.

15.3 Iray Bridge

Iray Bridge is a feature shipped with Iray that allows efficient use of the resources of a remote cluster to render a local scene over a relatively high-latency low-bandwidth connection such as the Internet. Iray Bridge is designed to be integrated with a minimal effort and to be easy to use.

Iray Bridge comes in the form of two plugins, one client-side plugin and one server-side plugin. The client-side plugin makes new Iray Bridge render modes available that can be used to render using a remote cluster the same way as when rendering locally. Swapping between local render modes and Bridge render modes can be done at any time. The

server-side plugin makes available the server-side Iray Bridge application that provides the server-side features needed by the Iray Bridge render contexts.

The data required on the remote host will be sent automatically by Iray Bridge as part of rendering a frame and features such as server-side caching of data and sending only what has changed since the last render minimize the data that needs to be sent. Iray Bridge supports streaming and low-latency hardware H.264 encoding and decoding to ensure high interactivity, as well as lossless video formats to deliver uncompressed frames for final renderings.

Iray Bridge also provides a scene snapshot feature that very efficiently saves the current state of the scene in a snapshot on the server that can later be imported off-line to perform things like batch renderings.

15.3.1 Network architecture

Iray Bridge is designed to use a client-server architecture. On the client side is an application with Iray integrated, for instance a CAD tool. The client can render locally as normal but can at any time switch to an Iray Bridge render context, which will automatically connect to the configured Iray Bridge server and use the resources of the server to render the scene. Any required data upload and download of rendered images will automatically be taken care of by Iray Bridge.

The server runs the Iray Bridge application and can be a single host or part of a cluster. Use cases range from connecting to the work station at the office when working remotely, a company owned cluster in a remote location, or to a virtual cluster rented by the hour. The Iray Bridge client only connects to a single server that needs to run the Iray Bridge application. The client will automatically get all the resources of any cluster that the Iray Bridge server is a part of. Note that the other hosts form an Iray cluster with the Iray Bridge server and these hosts doesn't need to run the Iray Bridge application. Also note that the Iray Bridge client doesn't form a cluster with the remote server, and also that the Iray Bridge client never participates in rendering when an Iray Bridge render mode is used.

The Iray Bridge client and Iray Bridge server use standard web sockets to communicate which gives the same performance as plain TCP but is accepted by many firewalls, making connections through corporate networks easier. The Iray Bridge application running on the server will listen on a web socket URL such as `ws://some.host.com/my_bridge_application` where some.host.com is the host name of the server and `my_bridge_application` is the name of the Iray Bridge application running on the server.

The Iray Bridge server can service multiple clients at the same time. The performance considerations are the same as starting multiple renderings on a local Iray host without using Iray Bridge.

15.3.2 Client integration

The Iray Bridge client-side features are made available by loading the plugin iray_bridge_client.dll/so. This plugin will make the Iray Bridge render modes available and install the API component: `mi::bridge::IIray_bridge_client` The only configuration needed is the web socket URL to the Iray Bridge application and, optionally, a security token if the Iray Bridge application on the server is set up to require that. In that case, the server application will inspect the security token and can reject the connection if it was not correct.

The application address is set by looking up the API component `mi::bridge::IIray_bridge_client` and calling: `mi::bridge::IIray_bridge_client::set_application_url`(const char* url) The URL is a web socket URL and both plain and secure web sockets are supported, for example ws://some.host/some_app or wss://some_host/some_app.

The optional security token is set by calling: `mi::bridge::IIray_bridge_client::set_security_token`(const char* token) These two methods set the default values that will be used by all the Iray Bridge render contexts when a connection to the Iray Bridge server needs to be established. The values can be changed at any time which will cause any active Iray Bridge render contexts to reconnect using the new values on the next rendered frame. It is also possible to override the Bridge application address and security token per Iray Bridge render context by setting the render context options `bridge_address` and `bridge_security_token`.

The Iray Bridge API component also allows creation of a snapshot context which works analogous to render contexts but can be used to save a very efficient snapshot of a scene on the server. The snapshot can later be imported off-line on the server, just like a regular scene, to provide features like batch rendering. The snapshot contexts also use the default Iray Bridge application address and security token configured on the `mi::bridge::IIray_bridge_client` API component and can also be overridden per snapshot context by setting the options `bridge_address` and `bridge_security_token`.

Iray Bridge can use hardware decoded H.264 for streaming images from the server. This feature is provided by loading the nvcuvid_video_decoder.dll/so plugin on the client. This plugin provides hardware H.264 decoding on NVIDIA GPUs that supports this. Note that the maximum resolution that is supported varies depending on the capabilities of the GPU. A software H.264 decoder fall-back can be provided by loading the ffmpeg_video_decoder.dll/so. The software decoder will automatically be used if the NVIDIA hardware decoder fails, for instance because no suitable NVIDIA GPU is installed, or because the resolution is too high.

Alternatively, Iray Bridge can also use any image format supported by Iray for streaming images from the server. Loading the OpenImageIO plugin provides a large number of image formats that suits different needs and have different size/quality trade-offs such as `jpg`, `png`, or `exr`. Loading other image plugins will automatically make the format available for use by Iray Bridge.

15.3.3 Server integration

The server side Iray Bridge features needed to back up the Iray Bridge render and snapshot contexts are made available by loading the plugin iray_bridge_server.dll/so. This plugin installs the API component: `mi::bridge::IIray_bridge_server` This API component can be used to create an instance of the Iray Bridge Application by calling: `mi::bridge::IIray_bridge_server::create_application`(const char* application_path, http::IServer* http_server) Iray Bridge uses web sockets to communicate between client and server, so before creating an application an HTTP server must be created and set up to listen to some interface and port. The create_application() method then creates an Iray Bridge application listening for clients on the provided path on the provided HTTP server. So if the HTTP server is set up to listen for HTTP requests on the standard port 80 on host some.host.com and the application is created with the path `some_app`, then the client would connect to it using the web socket URL ws://some.host.com/some_app.

Before the application can accept sessions it must be configured and opened for connections. The only mandatory configuration is the path to the disk cache which is set by calling: `mi::bridge::IIray_bridge_application::set_disk_cache`(const char* location) The disk cache is where data is cached to avoid having to upload the same data more than once. It is a persistent storage and is either a local path or a cache server. Prefixes are used to distinguish the two cases: “path:” for local path and “address:” to indicate an TCP/IP address to a cache server. Examples: `path:c:\temp\disk_cache` or `address:127.0.0.1:7777`.

If snapshots will be saved then a path where they are saved needs to be specified by calling: `mi::bridge::IIray_bridge_application::set_snapshot_path`(const char path) It is also possible to register a session handler by calling: `mi::bridge::IIray_bridge_application::set_session_handler`(IApplication_session_handler* handler) This optional handler will be called during the client handshake and can be used to inspect the security token provided by connecting clients and reject connections based on the token, or other things like restricting the number of concurrent sessions.

When configuration is finished, open the application for client sessions by calling: `mi::bridge::IIray_bridge_application::open`() The Iray Bridge application is now ready to accept client sessions and render on behalf of Iray Bridge render contexts.

Iray Bridge supports hardware H.264 encoding. This is enabled by loading the plugin nvenc_video.dll/so on the server. Note that a Kepler or newer NVIDIA GPU is required for hardware video encoding to work. Iray Bridge also supports software H.264 encoding by loading the plugin x264_video.dll/so. If both plugins are loaded then hardware H.264 encoding will be used if there is a GPU on the server that supports it, otherwise software encoding will be used.

> **Note:** These plugins are not shipped with Iray because of H.264 licensing considerations. For more information, contact NVIDIA ARC.

Iray Bridge also supports encoding using any image format supported by Iray. Loading the OpenImageIO plugin provides a large number of image formats that suit different needs and have different size/quality trade-offs such as `jpg`, `png`, or `exr`. Loading other image plugins will automatically make the format available for use by Iray Bridge.

15.3.4 Render modes

Loading the `iray_bridge_client` plugin makes three new Iray Bridge render modes available:

- iray_cloud (Photoreal Cloud)
- irt_cloud (Interactive Cloud)
- nitro_cloud (IQ Cloud)

These render modes can essentially be seen as proxies for server side counterparts of these render modes:

- iray (Photoreal)
- irt (Interactive)
- nitro (IQ)

Simply render with any of the Iray Bridge render contexts to automatically connect to the configured server, upload any data not already on the server, and use its resources for rendering.

> **Note:** Iray Bridge render modes have their own set of options and, generally, do not support the same options as their local counterparts. See "Render context options" (page 158) for a list of supported options.

15.3.5 Upload progress

The Iray Bridge render contexts automatically upload scene data as needed. Sometimes, especially when rendering the first frame of a new scene, this will upload a considerable amount of data which can take a long time. Provided a `IProgress_callback` instance was passed to the render call, the Iray Bridge render contexts will issue progress callbacks while rendering to indicate that an upload is taking place and report the current progress of the upload.

The following progress areas will be issued:

`bridge_upload_state`
: `0` – Detecting scene changes. Not yet known how many scene elements need updating.

  `1` – Calculating hashes. Calculating hashes for the data that needs updating.

  `2` – Querying server side cache status. Not yet known how much data needs to be uploaded to the server.

  `3` – Uploading. It is now known how much data needs to be uploaded.

  `4` – Pending. Waiting for server to finish processing the uploaded elements.

  `5` – Done. Upload is done.

`bridge_total_bytes_to_upload`
: Total number of uncompressed bytes to upload. In state `0` to `2` the number indicates the amount of data for the cache misses reported by the server so far.

`bridge_bytes_uploaded`
: Total number of uncompressed bytes uploaded so far for this upload.

`bridge_updated_elements`
: The number of scene elements that are being updated. In state `0` the number indicates the number of changed elements detected so far.

`bridge_pending_hash_calculations`
: The number of pending hash calculations that needs to be done.

`bridge_pending_cache_status`
: The number of cache status requests sent to the server for which no answer has arrived yet.

`bridge_pending_data_serialization`
: The number of cache misses that still require the data to be serialized before it can be sent.

`bridge_uploaded_element`
: The name and size in bytes of the currently uploaded element.

`bridge_uploaded_element_bytes`
: The number of bytes uploaded for the currently uploaded element

The most important area is the `"bridge_upload_state"` which will be fired as soon as rendering is stalled because of an upload. Iray Bridge will start by detecting what changes need to be uploaded since the last render. As soon as it has detected changes, it will start querying the server for cache status. If the server reports cache misses, then the client will start uploading the data. All this happens in parallel and the different states indicate when the client is done with a certain task, at which point some values will be final.

- In state `0` changes are being detected, so it is not known yet how many elements or how much data will be uploaded.
- In state `1` it is known how many elements needs to be updated and hashes for those elements are being calculated. The hashes are needed to be able to query the server for cache misses.
- In state `2` hashes for all updated elements have been calculated, but the server has not yet responded to cache status for all elements so the total amount of data to upload is not yet known.
- In state `3` it is known what needs to be uploaded.
- In state `4` the upload is done but the client is waiting for a confirmation that all changes have been persisted in the server side database.
- In state `5` the upload is completely done.

The rest of the areas give detailed information about the upload progress and give the values known so far or the final value depending on the upload state. The states are entered strictly in order `0-4`, but states may be skipped if they are already done or if there was nothing to do in the state.

15.3.6 Video modes

Iray Bridge is designed to work over relatively high-latency, low-bandwidth connections like the Internet. Because of this, there are different trade-offs that have to be made, depending on the use case. They cause Iray Bridge render modes to behave a little bit different than the local render modes.

There are two basic use cases that Iray Bridge supports: interactive and final rendering, and three render modes to back them up: `"interactive"`, `"synchronous"`, and `"batch"`, which are controlled by setting the Iray Bridge video context option `"scheduler_mode"`.

15.3.6.1 Interactive Scheduler Modes

The `"interactive"` and `"synchronous"` scheduler modes are designed for the interactive use case and support low-latency hardware H.264 encoding and decoding to make the most of the available bandwidth. These two render modes operate using a server-side render loop which is updated with client-side changes when calling `render` on the Iray Bridge render context. The server-side render loop will keep rendering in the background and picks up changes from the client, renders, and streams back images with the changes or with more iterations back to the client on a video stream.

The "synchronous" scheduler mode works exactly like local rendering in that calling render with a transaction will return a canvas containing a rendering of the scene in the state of that transaction. This is required when for instance blending the result with other render contexts or if the resulting rendered frame must contain the changes in the passed in transaction for some other reason. While sometimes required, this comes at a potentially big loss of performance, especially for high latency connections. The reason for this is that the frame rate is not only dependent on how fast the server can render a frame, but also on the round trip time to the server, the time it takes to encode and decode the frame, and the time it takes to send the frame over the network. For a connection to a server with a round trip of 100 ms the frame rate is capped at 10 fps, even without considering any other cost.

The "interactive" scheduler mode is designed to overcome this limitation by allowing the returned canvas to lag a little bit behind. Each call to the render method of the Iray Bridge render context will send any scene changes to the server, but instead of waiting for a frame that contains those changes the most recent frame from the server will be returned immediately. The server-side render loop constantly renders and streams frames to the client and will eventually pick up the changes. The client application needs to keep calling the render method of the Iray Bridge render context to keep streaming changes to the server and to display the most recent frame from the server. This works very well for interactive use cases and, together will hardware H.264 encoding and decoding, can achieve frame rates very close to the actual frame rate on the server if network conditions are good enough. The "interactive" scheduler mode is not sensitive to network lag, but the bandwidth must be high enough to support the bit-rate of the video stream. Another factor that can cause performance problems is if the network tends to buffer data and deliver it in bursts.

The "interactive" and "synchronous" scheduler modes also support other video formats such as jpg, as well as lossless formats like png and exr, but these image formats are very expensive to encode and decode and also use a lot more bandwidth than H.264, so for the best interactive experience, H.264 is recommended. It is possible to switch at any time between the supported scheduler modes so interactive can be used until a frame is needed that is guaranteed to be up-to-date at which point "synchronous" can be used for that frame, and then switch back to "interactive".

Note: In both these modes, the server will keep rendering iterations in the background, also when video frames are sent over the network, so convergence time is not affected by the video frame rate.

The interactive scheduler modes send frames over a video stream that only encodes a single canvas at a time making it impractical to separate out different aspects of the light in the same rendering as can be done using path expressions (LPEs). Simultaneous rendering of multiple canvases is, however, supported by the "batch" scheduler mode.

15.3.6.2 Batch Scheduler Mode

The "batch" scheduler mode is intended to be used for final renderings with no or infrequent updates on the client before the final image is ready. This mode always uses a lossless video format for transmission of rendered frames. In contrast to the interactive scheduler modes, the batch mode does not use Iray Bridge video streaming and does support multiple canvases rendered in one pass. This mode is therefore ideal for use together with light path expressions (LPEs).

15.3.7 Render context options

There are a number of render context options that are available for all Iray Bridge render contexts. These render context options control various aspects of the video streaming and which server to connect to.

There are also some options that are only available to specific Iray Bridge render contexts.

Options are set by calling `set_option()` on one of the Iray Bridge render contexts and passing the name of the context as a string and an `mi::IData` instance of the correct type as a value.

15.3.7.1 Server configuration options

`mi::IString bridge_address = ""`
: The address of the Iray Bridge application to connect to. Empty by default. If set this value will override the value configured by using the `mi::bridge::IIray_bridge_client` API component.

`mi::IString bridge_security_token = ""`
: The security token to use when connecting to the Iray Bridge application. Empty by default. If set this value will override the value configured by using the `mi::bridge::IIray_bridge_client` API component.

15.3.7.2 Video streaming options

The video streaming options control various aspects of how video frames are transmitted from the server to the client. The default values are tuned to give good interactive performance with decent quality over a fairly modest Internet connection, but may need to be tuned to give better performance, or quality, or both, depending on the network performance and use case.

`mi::IDynamic_array video_formats_interactive = ["h264"`
: The video stream formats to use per canvas of the render target used when rendering with scheduler mode set to `"interactive"` or `"synchronous"`. Each entry in the dynamic array must be of type `mi::IString` and specifies the format to use for all layers of the canvas with the same index in the render target. Default is an array with the single entry `"h264"`, but this can be set to any image format supported by Iray, like `jpg`, `png`, `exr`, and so on. If video_formats_interactive lacks an entry for a canvas a default will be used depending on the pixel type of the canvas.

  Note:

  - The different image formats will affect the precision (floating point to 8-bit truncation, loss of alpha channel, and so on) of the resulting canvas on the client side and the lossy formats will also degrade quality. Most formats are very expensive to encode and decode and will use considerably more bandwidth than H.264.
  - Iray Bridge also supports the format `"lossless"` which performs a lossless compression of the original canvas without loosing any information.

  This option is ignored if `scheduler_mode` is set to `"batch"`.

`mi::IDynamic_array video_formats_batch = [`

The video stream formats to use per canvas of the render target used when rendering with scheduler mode set to `"batch"`. Each entry in the dynamic array must be of type `mi::IString` and specifies the format to use for all layers of the canvas with the same index in the render target. Default is an empty array, but this can be set to any image format supported by Iray, like `jpg`, `png`, `exr`, and so on. If video_formats_batch lacks an entry for a canvas a default will be used depending on the pixel type of the canvas.

Note:

- The different image formats will affect the precision (floating point to 8-bit truncation, loss of alpha channel, and so on) of the resulting canvas on the client side and the lossy formats will also degrade quality. Most formats are very expensive to encode and decode and will use considerably more bandwidth than H.264.
- Iray Bridge also supports the format `"lossless"` which performs a lossless compression of the original canvas without loosing any information.

This option is ignored if `scheduler_mode` is set to `"interactive"` or `"synchronous"`.

`mi::IString video_format = "h264"`

This is a convenience option that is equivalent to setting the option video_formats_interactive with an array with a single entry with the value of this option. Getting this option will return the value of index 0 of the video_formats_interactive option.

This option is ignored if `scheduler_mode` is set to `"batch"`.

`mi::IUint32 video_framerate = 30`

Sets the maximum video frame rate. The server will not send out frames faster than the specified setting, but will allow the frame rate to drop if the server-side render loop is slower than the set frame rate. If H.264 is used as a video format, this setting will be used together with the bit rate to calculate the budget for video frames. Bridge will however automatically make use of the set bit rate by improving video quality if the actual frame rate is lower than the set maximum frame rate. This automatic adaption is true only when video format is set to `"h264"`.

This setting has no effect if video mode is set to `"batch"`.

`mi::IUint32 video_min_bitrate = 1000000`

This is the minimum bit rate Bridge will use when encoding the video stream.

Bridge will automatically adjust the actual bit rate for the video stream depending on the estimated available bandwidth of the connection, but only within the span of the configured minimum and maximum bit rate. Setting the minimum and maximum bit rate to the same value will force Bridge to use that bit rate regardless of the available bandwidth. If less bandwidth is available than the minimum bit rate setting then Bridge will automatically lower the frame rate to compensate. Bridge will never use more than the maximum bit rate setting, even if this is lower than the configured minimum bit rate.

Only used if the video format is set to `"h264"`, in which case, the H.264 encoder will be configured to use at least this bit rate.

`mi::IUint32 video_max_bitrate = 150000000`

This is the maximum bit rate Bridge will use when encoding the video stream.

Bridge will automatically adjust the actual bit rate for the video stream depending on the estimated available bandwidth of the connection, but only within the span of the configured minimum and maximum bit rate. Setting the minimum and maximum bit rate to the same value will force Bridge to use that bit rate regardless of the available bandwidth. If less bandwidth is available than the minimum bit rate setting then Bridge will automatically lower the frame rate to compensate. Bridge will never use more than the maximum bit rate setting, even if this is lower than the configured minimum bit rate.

Only used if the video format is set to `"h264"`, in which case, the H.264 encoder will be configured to use at most this bit rate.

`mi::IString video_tag = ""`

Use this tag with interactive scheduler mode when the client changes something in the scene and wants to know, without switching to synchronous mode, when a frame arrives with this change.

If the tag is set, then all subsequent calls to render() send this tag to the server, together with any client-side changes. Just before the render() call returns, a progress callback is fired with the area video_tag. The string value is the user-set tag that represents the changes the frame contains. If the tag is empty, then no progress callback is made.

> **Note:** If the application sets the video tag to `"1"`, `"2"`, `"3"` without waiting for frames with previous tags, then tags might be skipped. In this example, only tag `"3"` might be returned. Tags will, however, always arrive in order.

`mi::ISint32 nvcuvid_device = -1`

The index of the device to use on the client for hardware H.264 decoding. If set to -1 (the default), a suitable device will be selected automatically.

> **Note:**
>
> - Only NVIDIA GPUs that support hardware H.264 decoding can be used. The resolution limit depends on the GPU. Higher resolutions than the card supports will cause decoding to fail.
> - The GPU must not be in TCC (Tesla Compute Cluster) driver mode. H.264 hardware decoding does not work in this mode.

`mi::IFloat32 idle_timeout = 1.0`

The time in seconds for which the server side render loop will keep rendering in the background without any render requests coming from the client. Defaults to 1 second.

`mi::IString scheduler_mode = "interactive"`

The scheduler mode controls if rendering is optimized for interactivity or for final renderings. Can be set to `"interactive"`, `"synchronous"` or `"batch"`. Should be set to `"interactive"` for interactive use cases and `"batch"` for final renderings. The `"synchronous"` mode can be used for interactive use cases where it is important that the rendered canvas match the render transaction, but at a usually very large performance penalty. The interactive scheduler mode is optimized for maximum interactivity and can give frame rates very close to the frame rate on the server if network conditions are good enough, even when latency is high. It does this by allowing frames to lag behind, so

generally frames returned by a call to render does not include the changes in the render transaction.

15.3.7.3 Photoreal Cloud render mode options

These options are specific to the `"iray_cloud"` render context:

`mi::IFloat32 batch_update_interval = 1e36`
How often a progressive image update is rendered when in batch mode. Ignored when `scheduler_mode` is set to `"interactive"` or `"synchronous"`.

15.3.7.4 IQ Cloud render mode options

These options are specific to the `"nitro_cloud"` render context:

`mi::IString scheduler_mode = "interactive"`
In addition to the scheduler modes `"interactive"`, `"synchronous"`, and `"batch"`, it is also possible to set the scheduler mode to `"streaming"`.

The `"streaming"` scheduler mode improves the frame rate by introducing a small lag, in addition to the lag introduced by the interactive mode, which allows rendering to continue while the previous frame is composited on the server, but otherwise works the same way.

`mi::IFloat32 batch_update_interval = 1e36`
How often a progressive image update is rendered when in batch mode. Ignored when `scheduler_mode` is set to something other than `"batch"`.

`mi::IUint32 min_samples_per_update = 1`
The minimum number of samples per update. Each frame will contain at least this many iterations. Defaults to `1`.

`mi::IUint32 max_samples_per_update = 0`
The maximum number of samples per update. Each frame will contain at the most this many iterations. Defaults to `0`, which means unlimited. The render context will automatically increase the number of iterations per frame over time when there are no changes to the scene to increase efficiency. Setting this to, for instance, `1` means that there will be one rendered frame per iteration. Setting this to `0` can mean that after rendering for a while, restarting interactive usage can take a long time.

15.3.7.5 Interactive Cloud render mode options

The render mode `"irt_cloud"` currently does not have any render mode specific options.

15.3.8 Server-side data cache

Any scene data sent to the Iray Bridge server will be persistently stored on the server to avoid sending it several times. The client will only send hashes of the data and the server will request data only for hashes it has not already stored in the cache. This greatly reduces the amount of data that needs to be uploaded if rendering the same scene several times in different sessions, or if, for instance, large textures are shared between multiple scenes.

The cache is either saved to a folder on a disk drive local to the server, or to a cache server. Cache servers can be used in certain Iray Bridge configurations to make cached data available

to multiple hosts, not only the host it was saved to. This is very useful in a scenario where users can reserve nodes from a big cluster of machines but might not get the same Iray Bridge head node each time, in which case the cached data from the previous session will be distributed and available to all nodes.

The cache to use is specified by creating an Iray Bridge application and calling `mi::bridge::IIray_bridge_application::set_disk_cache`(const char* location). The location string is either prefixed with “address:” followed by a TCP address of an already running cache server, or “path:” followed by a local folder path.

Currently, all data uploaded to the Iray Bridge server will be stored permanently in the cache, so it will grow indefinitely as more and more data is sent to the server. The disk cache folder together with all the generated files in it can however be removed when not in use to start with a clean cache.

15.3.9 Proxy scene element

Iray Bridge supports a Proxy scene element that can be attached to Group or Instance elements. On the client, a Proxy scene element only contains a reference ID to a sub-scene on the server-side. The sub-scene is, in turn, represented by a Bridge snapshot that is generated beforehand.

Using a Proxy scene element to represent a sub-scene is useful when you want to support lower-quality rendering on the client side and minimize the number of sub-scenes that need to be re-rendered. High-quality rendering that incorporates snapshots of sub-scenes is limited to the server-side where it makes more sense.

Note:

- The name of the proxy element must be the same as the top-level element of the sub-scene that the proxy represents. There can only be one top-level element.
- The top level element of the sub-scene must be a legal attachment to the element that the Proxy is attache to on the client.
- Bridge snapshots of the sub-scenes must be made beforehand. See `mi::bridge::IBridge_server::create_snapshot_context`() for more information on creating snapshots.

15.3.10 Scene snapshots

In addition to cloud rendering, Iray Bridge also supports saving very efficient scene snapshots on the server. The snapshots are saved as very small files using the extension `.cb` and can be imported just like normal files from the server host where they were saved, as long as the `cb` importer plugin is loaded. Snapshots can be used for off-line batch rendering and can be either full or incremental. Incremental snapshots are ideal for batch rendering of animations since the import of each incremental snapshot will only update a few scene elements and then the next frame can be rendered without having to pay for the extra overhead that is incurred by loading and rendering a new scene.

Iray Bridge snapshots are created on the client by using a snapshot context. The snapshot context is designed to work analogous to an Iray Bridge render context, except that instead of rendering it will create snapshots on the server. The snapshot context will also automatically

upload any required data to the server, only sending data that has changed since the last saved snapshot and also only sending data that is not already available in the server-side data cache. The snapshot context fires the exact same progress areas as the Iray Bridge render contexts when uploading data.

Snapshot contexts are created by looking up the `mi::bridge::IIray_bridge_client` API component and calling create_snapshot_context() which returns a `mi::bridge::IIray_bridge_snapshot_context` instance. This instance will be tied to the scene and the scope of the transaction it was created with and snapshots can be created by calling create_snapshot().

Snapshot data is saved in the data cache configured for the Iray Bridge application on the server, and the snapshot file with the extension `.cb` is stored in the server-side folder configured for the Iray Bridge application. The `.cb` file contains the ID of the snapshot and the address or path to the data cache where the data is stored. The `.cb` file can normally only be imported from the same host where it was saved.

15.3.11 Example – Rendering a scene with Iray Bridge

Source code

`example_bridge_client.cpp` (page 198)
`example_bridge_server.cpp` (page 204)

Iray Bridge is implemented on top of the Bridge API, a generic API that is part of the Iray API. This API allows third-party applications that embed Iray to create custom server-side Bridge applications that can be accessed by custom Bridge clients. The Bridge API provides all the core features used by Iray Bridge, such as efficient data transfer, server-side caching of data, video streaming, and so on.

15.3.11.1 Element Data Override Render Context Option

Because Bridge allows data for an element to be different on the server and the client, you can use the `hash_override` option supported by Bridge render contexts to enable lower resolutions for meshes and textures on client devices and switch to higher resolutions when rendering on a VCA or high end computer running Iray Server.

`mi::IMap hash_override =`
: A Map<String> with element names as keys and hash overrides as values. The map contents must not be changed after setting this option. To change the overrides a new map needs to be created. Clear overrides by using a NULL map argument.

Apply the `hash_override` option to specific scene elements. When Bridge comes across an element where the hash override is specified, it skips normal serialization and hash calculation and sends the element to the server. The server loads the cached data specified by the hash and uses this data instead of the original data on the client.

Note:

- The data for the hash override must be stored in the server-side cache beforehand by creating a Bridge snapshot that contains the data. Cache misses for elements with hash overrides are errors. See `mi::bridge::IIray_bridge_client::create_snapshot_context()` or `mi::bridge::IBridge_server::create_snapshot_context()` for more information on creating snapshots.
- Obtain the override hash for an element by using the `mi::bridge::IBridge_server::calculate_hash()` method.
- You cannot change the element containing the hash override in any way. Changes to the element on the client are only visible on the client. Because the server always uses the hash override, the data on the server remains unchanged. However, you can change the hash override between render calls.
- The `hash_override` option is per scene, per scope, and per session, but not per render context. For example, imagine two render contexts — one `iray_cloud` and one `irt_cloud` — with the same scene, scope, and server address. If you set the `hash_override` option for one render context, both render contexts are affected. This is because the same Bridge session is shared for all render contexts that use the same server address.

16 Server-based rendering

The following sections describe how to deliver images rendered on a server to the client.

16.1 Example – Running an HTTP server

Source code
`example_http_server.cpp` (page 224)

This example provides a very simple HTTP server which serves one particular image. The image is rendered in the same way as previous examples such as "A first image" (page 8).

The `example_http_server.cpp` (page 224) program uses a simple request handler that always serves a fixed image, independent of the requested URL. A response handler is installed to set the content type to `image/jpeg`. This is just done for illustration purposes; the content type could also be set directly by the request handler.

> **Note:** For a general introduction to the HTTP server, see `HTTP server`.

16.2 RTMP server

The RTMP server implements the RTMP standard (see the RTMP specifications [RTMPProtocolChunkStream] (page 197) , [RTMPMessageFormats] (page 197) and [RTMPCommandMessages] (page 197)) and, much like the HTTP server, allows you to write applications based upon C++. However, unlike HTTP-based applications, RTMP-based applications are event based. The client sends asynchronous command messages to the server, and the server responds to these command messages through a set of functions registered to handle such command messages. This is facilitated by the stateful nature of RTMP and greatly simplifies application development.

Another difference between RTMP- and HTTP-based applications can be seen in how efficiently they use TCP connections. HTTP-based applications tend to have the client send a request, wait for the server's response, then send another request. Due to the cost of establishing TCP connections, TCP's slow start and the congestion avoidance algorithms used in most operating systems, this mode of TCP connection utilization does not lead to optimal usage of the bandwidth available. RTMP, on the other hand, is bidirectional and can multiplex several logical flows over one connection, including keep alive and other meta data packets, thus avoiding the HTTP request/response round-trip behavior and the creation of new TCP connections. As a result, RTMP based applications lead to a more optimal usage of the available bandwidth and thus a richer user experience.

Neuray needs a video encoding plugin to be able to do video streaming unless the data to be streamed out is already video encoded. A video streaming plugin is installed like any other plugin: They are loaded explicitly in neuray using the `IPlugin_configuration::load_plugin_library()` or

`IPlugin_configuration::load_plugins_from_directory()` methods. The latter allows you to offer automatic plugin loading from a specified directory.

Currently neuray comes with one bundled video plugin. This uses the so called "Flash screen video" format. It is a very simple format which can be played by the Flash client. The format provides only a very small reduction in the needed bandwidth and thus should only be used when the network between the server and the clients provides very high bandwidths, for example, for testing in a LAN.

Other video plugins providing more sophisticated formats may be made available as separate downloads.

16.3 RTMP client

The RTMP server supports any RTMP client, the industry standard being the one integrated into Adobe Flash/Flex. The method of connecting to and displaying an RTMP stream from neuray is identical to connecting to any other RTMP server. Adobe Flex 3 will be the example platform used; however, the concepts will transfer to any other platform.

The most convenient way to integrate RTMP stream into a Flex application is to create a new `UIComponent` which can simply be placed onto your Flex application canvas. This component will need instances of the three standard classes required to stream RTMP video: NetConnection, `NetStream` and Video.

```
public class Rs_rtmp_video extends UIComponent {
    private var m_connection:NetConnection = null;
    private var m_video:Video = null;
    private var m_stream:NetStream = null;
    ...
}
```

The first thing that is required is to create a connection to the neuray RTMP server in an init() function.

```
public function init(url:String):void
{
    m_connection = new NetConnection();
    m_connection.addEventListener(NetStatusEvent.NET_STATUS, ↦
   net_status_handler);
    m_connection.connect(url);
}
```

When the connection is complete (or fails), Flex will call the `net_status_handler` listener registered in the init() call. This handler can then create the `NetStream` object and begin displaying the video.

```
private function net_status_handler(event:NetStatusEvent):void
{
  if (event.info) {
    switch (event.info.code) {
      case "NetConnection.Connect.Success":
```

```
            m_stream = new NetStream(m_connection);
            m_video = new Video(this.width,this.height);
            m_video.attachNetStream(m_stream);
            addChild(m_video);
            m_stream.play("rs_stream");
            break;
        case "NetConnection.Connect.Failed":
            break;
    }
  }
}
```

The above implements a fully functional RTMP video streaming client. However, it is not particularly useful as it provides no interaction with neuray. For this we need to use the NetConnection::call interface to call registered RTMP call command message handlers.

The registered call handler is then called via the `NetConnection` call mechanism:

```
protected function move_camera(pan_x:Number, pan_y:Number):void
{
     var arguments:Object = new Object();
     arguments["pan_x"] = pan_x;
     arguments["pan_y"] = pan_y;
     m_connection.call("move_camera",null,arguments);
}
```

The `move_camera` function would typically be called in reaction to mouse movement events.

The server side C++ application handler code for the above command call would look something like the code snippet below and would be registered on the server side as a call command handler using the Iray API method `mi::rtmp::IConnection::register_remote_call_handler()` on the `mi::rtmp::IConnection` interface.

```
class Call_event_handler : public
    mi::base::Interface_implement<mi::rtmp::ICall_event_handler>
{
public:
    bool handle(
        const char* procedure_name,
        const mi::IData* command_arguments,
        const mi::IData* user_arguments,
        mi::IData** response_arguments)
    {
        // Get the x and y coordinates from the user_argument and
        // reposition the camera.
        ...
        return true;
    }
};
```

The examples included in the distribution include a complete flash client implementation with commented source code. This, together with the example RTMP application server source code provides a great start for building video streaming solutions.

Note: Some older Adobe Flash clients on Linux, for example 9.0.124, cannot show the delivered video frames fast enough which severely degrades the interactivity. The client version is provided as a command argument to the connect event handler in the field `flashVer` and it is recommended to check this variable and either lower the frame rate or recommend an Adobe Flash client upgrade to users with old Flash clients.

16.3.1 Example – Running an interactive video-streaming server

Source code

`example_rtmp_server.cpp` (page 285)
`example_rtmp_server.mxml` (page 293)
`example_rtmp_server_actionscript.as` (page 294)

The example programs render a scene and serves an interactive video stream over RTMP. The provided Flash file (the .swf file) can be reproduced running the free Adobe Flex SDK compiler `mxmlc` on the included .mxml file.

Note the following:

- The `example_rtmp_server.cpp` (page 285) starts a HTTP server. A Flash application is fetched with a browser or a standalone Flash client, which the client then uses to view and interact with the video stream.
- By default, the stream uses the screen video codec, which is provided with the library and encodes the canvas produced by scene rendering. The encoded frame is sent over the RTMP stream to the Flash client.
- On the connection, a Remote Procedure Call (RPC) is installed. The RPC is called when the client interacts with the video stream using the mouse.

17 Customization

Note that these customizations are rarely needed.

17.1 Extending the database with user-defined classes

This section introduces the important API interfaces and steps to extend the database for user-defined classes:

- Defining the interface of a user-defined class (page 169)
- Implementing a user-defined class (page 169)
- Example – A user-defined class (page 170)

The program `example_user_defined_classes.cpp` (page 323) demonstrates the implementation and usage of user-defined classes to be used in conjunction with the Iray API. In this example, a new class is defined in and used by the main application.

> **Note:** In contrast, the example for plugins, described in "Extending neuray with plugins" (page 170) demonstrates how to provide additional functionality with user-defined classes by using plugins.

A simple class called `My_class` defined in `my_class.h` (page 330) with a corresponding interface class `IMy_class` defined in `imy_class.h` (page 329) demonstrate custom class definition.

17.1.1 Defining the interface of a user-defined class

To define user-defined classes you need to provide an interface with the public methods of your class. The interface is an abstract base class which contains only pure virtual methods. The interface must be (directly or indirectly) derived from `IUser_class`.

The easiest way to define such an interface is by deriving from the mixin class `mi::base::Interface_declare`. The first eleven template parameters are used to construct the `mi::base::Uuid_t` (which must be unique). The last template parameter denotes the actual interface to derive from (should be `IUser_class`, or another interface (directly or indirectly) derived from `IUser_class`).

This definition of the interface is required by the applications (or other plugins) in which the user-defined class is supposed to be used.

17.1.2 Implementing a user-defined class

You need to implement the interface defined in `imy_class.h` (page 329).

Deriving your implementation from the mixin `User_class` is required to ensure that your implementation class works together with the class framework, including features like

creation, serialization and deserialization. Note that if you derive your implementation class directly from your interface or indirectly via `mi::base::Interface_implement` it will not correctly work together with the class framework.

To make serialization and deserialization work, you need to implement `ISerializable::serialize`() and `ISerializable::deserialize`(). In this example it is sufficient to write the single member `m_foo` to the serializer and to read it from the deserializer, respectively. Furthermore, you need to implement `IUser_class::copy`() which has to create a copy of a given object.

17.1.3 Example – A user-defined class

Source code

`imy_class.h` (page 329)
`my_class.h` (page 330)
`example_user_defined_classes.cpp` (page 323)

The example program demonstrates how to use the user-defined class. Before using the new class it is necessary to register it with the Iray API. This can be accomplished by `IExtension_api::register_class`. Using the user-defined class works the same way as for all other classes. You can use `ITransaction::create`() to create new instances; you can store them in the database and retrieve them from the database. Here, the example just does a simply test of the serialization and deserialization functionality.

17.2 Extending neuray with plugins

This section introduces important interfaces and the required steps to add functionality and package it as a plugin:

1. Implementing a plugin (page 170)
2. Loading a plugin (page 171)

The programs `plugin.cpp` (page 332) and `example_plugins.cpp` (page 257) demonstrate how to provide additional functionality as plugins. This example extends the user-defined classes described in "Extending the database with user-defined classes" (page 169). The functionality of the `IMy_class` interface is provided to the main application by using a plugin. Therefore, the main application only needs the definition of the interface `IMy_class` and the compiled plugin, but not the source code of the implementation.

17.2.1 Implementing a plugin

To implement a plugin you need to provide an implementation of the `IPlugin` interface. An example implementation of such a class is shown in the code below. The most important part is the implementation of the `IPlugin::init`() method which has to register all user-defined classes.

Finally you need to provide a factory function called mi_plugin_factory() that creates an instance of your implementation of `IPlugin`. Again, using the example implementation as shown in the code below is fine.

You need to compile the implementation of your interfaces, your implementation of `IPlugin`, and the factory function, and to create a DSO from the object code. Using g++ you can use commands similar to the following:

```
g++ -I$IRAY_ROOT/include -fPIC -c plugin.cpp -o plugin.o
g++ -shared -export-dynamic --whole-archive plugin.o -o libplugin.so
```

17.2.2 Loading a plugin

You can load plugins by passing the filename to the method `IPlugin_configuration::load_plugin_library()`. After the plugin has been loaded (in a way similar to the loading of the DSO for the Iray API itself), the method `IPlugin::init()` is called. In this method you need to register all user-defined classes.

Using the user-defined class works the same way as for all other classes. In particular there is no difference to user-defined classes defined in the main application as described in "Extending the database with user-defined classes" (page 169).

17.2.3 Example – Creating a plugin

Source code

`plugin.cpp` (page 332)
`example_plugins.cpp` (page 257)

This example shows how to add functionality to the API through the integration of a user-defined class.

17.3 Extending supported file formats with plugins

The section "Implementation of an importer" (page 172) describes the implementation and usage of custom importers to be used in conjunction with the Iray API. In this example the new importer is defined in and used by the main application for simplicity.

> **Note:** Like any other user-defined class, it is possible to provide custom importers by using plugins. For an example, see "Extending neuray with plugins" (page 170).

A simple importer called `Vanilla_importer` is used in the example program `example_importer.cpp` (page 228) to demonstrate the basic steps. This importer is an illustrative skeleton that implements all interfaces but does not actually parse the file content in a meaningful way.

The section "Implementation of an exporter" (page 173) demonstrates the implementation and usage of custom exporters to be used in conjunction with the Iray API. In this example the new exporter is defined in and used by the main application for simplicity.

> **Note:** Like any other user-defined class, it is possible to provide custom exporters by using plugins. For an example, see "Extending neuray with plugins" (page 170).

A simple exporter called `Vanilla_exporter` is used in the example program `example_exporter.cpp` (page 209) to demonstrate the basic steps. This exporter is an illustrative skeleton that implements all interfaces but does not actually write elements out; it just writes their types and names.

The section "Implementation of an exporter" (page 173) describes the export of rendered pixel data to disk. Although neuray supports a wide range of image formats you might want to add support for your own image format. The example program `example_psd_exporter.cpp` (page 268) exports images in the Photoshop PSD file format [PFFS10] (page 197). The example program demonstrates how to export several canvases with different content (rendered image, normals, z-buffer, etc.) into the same file.

17.3.1 Implementation of an importer

This section introduces the basic concepts about the implementation and usage of custom importers used in conjunction with the Iray API:

- Implementing an importer (page 172)
- Registering an importer (page 173)

An example importer program consisting of `vanilla_importer.h` (page 342) and `example_importer.cpp` (page 228) demonstrate these concepts. The importer is called Vanilla. For simplicity, it is defined in and used by the main application. The importer is intended as an illustrative skeleton — it implements all interfaces but does not parse the file content in a meaningful way.

17.3.1.1 Implementing an importer

The implementation of the Vanilla importer is structured in three parts:

- Implementing the `IImpexp_state` interface
- Implementing the `IImporter` interface
- Implementing the `IImporter::import_elements`() method

Instances of `IImpexp_state` are used to pass information about the current importer state, for example to recursive calls of importers. The Vanilla importer does not need to carry around any additional information besides what is required by the interface, therefore this simple implementation is fine. Note that the simple derivation from `mi::base::Interface_implement` suffices here (in contrast to `example_plugins.cpp` (page 257)).

The Vanilla importer is given in the implementation of the `IImporter` interface. Later, an instance of this class is registered as an importer with the Iray API. Most of the methods implemented here are defined in the base interface `IImpexp_state`, as they are common for importers and exporters. The Vanilla importer claims to handle files with the extension `.vnl` and `.van`. It does not require specific capabilities of the reader to handle these formats. However, if the reader supports the lookahead capability, it will use a magic header check instead of relying on file name extensions.

The actual work of the Vanilla importer occurs in the import_elements() method. It is split into three parts:

1. Creating an import result object
2. Creating a group object which acts as the root group
3. Reading the file line by line

While performing these tasks, the example program demonstrates what type of errors to detect, how errors can be reported, and how to implement an include file mechanism, and similar things, with a recursive call to the `IImport_api::import_elements`() method.

17.3.1.2 Registering an importer

Registering importers is similar to registering user-defined classes. However, since importers are different from regular classes (for example, you cannot create instances of them using `ITransaction::create`()); you need to use a registration method specific to importers. This registration method expects a pointer to an instance of the custom importer.

To run the example, you need to create two files called test1.vnl and test2.van, each five lines long and starting with a line saying `VANILLA`. For demonstration purposes, the example will print the generated error messages and the names of the imported elements.

17.3.1.3 Example – Creating an importer for a custom file format

Source code

`vanilla_importer.h` (page 342)
`example_importer.cpp` (page 228)

This example registers an importing process for a custom file format defined by classes in a header file. The file is read and the imported data are printed.

17.3.2 Implementation of an exporter

This section introduces the basic concepts about the implementation and usage of custom exporters used in conjunction with the Iray API:

- Implementing an exporter (page 173)
- Registering an exporter (page 174)

An example exporter program, consisting of `vanilla_exporter.h` (page 334) and `example_exporter.cpp` (page 209), demonstrate the use of these concepts. The exporter is called Vanilla. For simplicity, it is defined in and used by the main application.

17.3.2.1 Implementing an exporter

The implementation of the Vanilla exporter in the example source is structured in three parts:

1. Implementing the `IImpexp_state` interface
2. Implementing the `IExporter` interface
3. Implementing the `IExporter::export_scene`() and the `IExporter::export_elements`() methods

Instances of `IImpexp_state` are used to pass information about the current exporter state, for example to recursive calls of exporters. The Vanilla exporter does not need to carry around any additional information besides what is required by the interface, therefore this simple implementation is fine. With the exception of the element list flag (which is not needed for exporters), it is the same implementation as for the Vanilla importer. Note that the simple derivation from `mi::base::Interface_implement` suffices here (in contrast to `example_plugins.cpp` (page 257)).

The Vanilla exporter is given in the implementation of the `IExporter` interface. Later, an instance of this class will be registered as exporter with the Iray API. Most of the methods implemented here are actually defined in the base interface `IImpexp_state`, as they are common for importers and exporters. The Vanilla exporter claims to handle files with the extension `.vnl` and `.van`. It does not require specific capabilities of the writer to handle these formats.

The actual work of the Vanilla exporter occurs in the export_scene() and export_elements() methods. It is split into three parts:

1. Creating the export result object
2. Setting up some data structures
3. Traversing the scene graph and writing the file, element by element

The map is used to perform the depth-first traversal of the scene graphs. As an example, the loop expands elements of type `IGroup` and `IInstance` and follows to the elements mentioned as their items. Other scene elements that need traversal are not handled in this example.

While performing these tasks the example demonstrates what type of errors to detect, how errors can be reported, and how to implement an depth-first traversal of the scene graph.

The export_elements() member function uses the same code fragments as the export_scene() member function above. In its overall structure, the export_elements() member function is just simpler in that it does not need to recursively follow any elements. It just exports all elements given in its parameter.

17.3.2.2 Registering an exporter

Registering exporters is similar to registering user-defined classes. However, since exporters are different from regular classes (for example, you cannot create instances of them using `ITransaction::create`()); you need to use a registration method specific to exporters. This registration method expects a pointer to an instance of the custom exporter.

To run the example, you need to call it with an existing scene file for import. The exporter will create a file called test3.vnl, which contains the types and names of all elements in the scene.

17.3.2.3 Example – Creating an exporter for a custom file format

Source code

`vanilla_exporter.h` (page 334)
`example_exporter.cpp` (page 209)

This example demonstrates the implementation of a importer for a custom file format. The filename and the MDL search path are provided as command-line arguments.

17.3.3 Example – Creating an image exporter for Photoshop

Source code

`example_psd_exporter.cpp` (page 268)

This example exports images in the Photoshop PSD file format PFFS10. The example program demonstrates how to export several canvases with different types of scene data (rendered image, normals, z-buffer, etc.) into the same file.

17.3.3.1 Implementing an image exporter

There are two ways to add support for an image format:

- By using an image plugin that adds support for this file format. The image plugin has the advantage that it extends the generic export facilities with the new image format.
- By using an explicit method that does the job. the explicit method has the advantage that you have more freedom and are not bound to the framework of image plugins.

In this example, the goal is to:

1. Export multiple canvases into one file
2. Export the names of the canvases

An explicit method is used because the framework for image plugins does not support these two features.

The example code is structured as follows:

- Auxilliary methods (from `write_int8` to get_pixel_type()) that deal with low-level export of certain data formats and pixel type properties
- The main method export_psd() that accepts an array of canvases to export
- An adaptor of this main method with the same name that accepts an instance of `IRender_target` to export
- An implementation of the `IRender_target` interface with three canvases that will be used in this example
- The example code to set up and to render a scene (essentially the same as in `example_rendering.cpp` (page 282)).

While the implementation of an image exporter heavily depends on the image format in question, there are often similarities in the file format structure and in the tasks to be accomplished. Typically, there is some sort of file header with meta information, followed by the actual pixel data.

To export a PSD file, the example program iterates over all the canvases to compute the needed metadata (width, height, number of layers, bits per channel) and reject invalid inputs. Next, the file is opened and the file header is written.

The next section in the PSD file format is the layer and mask information section. This section is split into two parts:

- The first part contains all the layer records which are basically a header with metadata for each layer, including the name of the layer.
- The second part contains the actual pixel data for each layer (in the same order as described by the layer records).

Finally, there is a last section named image data section. This section is supposed to contain the pixel data of all layers merged. It is primarily needed for applications that do not deal with the layer and mask information section. In this example, the pixel data of the first canvas is exported again (for simplicity).

When writing your own image exporter, or importer, there are several conversion tasks you have to handle:

1. Mapping neuray pixel types (see Types) to pixel types of the image format and/or vice versa.
2. Conversion of pixel data to a different pixel type (if needed).
3. Flipping the scanline order (if needed). In neuray, scanlines are ordered bottom-up.
4. Different tiling of pixel data.

The methods `IImage_api::read_raw_pixels`() and `IImage_api::write_raw_pixels`() can be very handy for the last three tasks. If requested, they can convert the pixel type as needed and flip the scanline order. They also convert the data between a possibly tiled canvas and a plain memory buffer.

17.3.3.2 Using the PSD example exporter

To demonstrate the PSD exporter, the example code is similar in its overall structure to the program in "Example – Importing and rendering a scene file" (page 10). To demonstrate the export of multiple canvases, a different implementation of the `IRender_target` interface is used. This implementation holds three canvases:

- One for the rendered image
- One for the normals
- One for the z-buffer

An additional method normalize() modifies the values for the normals and z-buffer such that the data is in the range [0,1] expected by most image formats.

After the image has been rendered, the render target is passed to the method export_psd(), which exports all canvases of the render target into the given file. For comparison, the canvases are additionally exported into individual PNG files.

18 Reference

The following sections provide detailed information about supported file formats and URI schemes, canvas names in render targets, and light path expression grammar.

18.1 Supported file formats and URI schemes

The following sections describe supported URI schemes, file formats for scene files and images, as well as the binary storage format.

18.1.1 URI schemes

All import and export methods use a common naming convention to identify files based on URIs as defined in [RFC3986] (page 197). Currently only the URI scheme `file` is supported. An empty URI scheme defaults to the `file` scheme. URI authorities are not supported, except for the empty URI authority, which is needed in some cases to resolve ambiguities (see below).

Within the `file` scheme relative and absolute URI paths are supported (see Section 4 in [RFC3986] (page 197)) URI paths are mapped to file system paths as follows.

- On Linux and Mac OS, URI paths are treated as file system paths without any translation.
- On Windows, slashes in relative URI paths are replaced by backslashes to obtain the file system path.
- On Windows, absolute URI paths are mapped to file system paths according to the following table:

URI path	*File system path*	*Comment*
`/C:/dir1/dir2/file`	`C:\dir1\dir2\file`	
`/C/dir1/dir2/file`	`C:\dir1\dir2\file`	This mapping is supported in addition to the first one since a colon is a reserved character in URIs.
`/dir1/dir2/file`	`\dir1\dir2\file`	This mapping is only supported for top-level directory names not consisting of a single letter.
`//share/dir1/dir2/file`	`\\share\dir1\dir2\file`	This mapping requires an (otherwise optional) empty URI authority (`//`) since otherwise the share name is interpreted as URI authority.

Unless the `{shader}` variable, explained below, is used, the relative path is resolved relative to the current working directory of the application, or for recursive imports and exports, relative to the scene file referencing it.

The `{shader}` variable can be used in URI paths, which is expanded by neuray. The URI path may start with the string `{shader}` followed by a slash and a relative path, in which case the relative path is resolved relative to the site's shaders directory. If several shaders directories have been configured, they are searched in turn. The `{shader}` variable is only supported for import methods, not for export methods. It is primarily intended to be used for MDL modules (`.mdl` files).

18.1.2 Scene file formats

neuray has built-in support for importing scene data from the following file formats:

- mental images binary scene format `.mib`, see "Binary storage format" (page 181)
- Material Definition Language (MDL) materials `.mdl`, see "Material Definition Language" (page 90) and [MDLSpecification] (page 197)

neuray has built-in support for exporting scene data to the following file formats:

- mental images binary scene format `.mib`, see "Binary storage format" (page 181)

In addition to that the neuray installation comes with plugins to support both the import and export of the mental images scene format `.mi`, see [Driemeyer05] (page 197). These plugins need to be loaded explicitly in neuray using the `IPlugin_configuration::load_plugin_library`() or `IPlugin_configuration::load_plugins_from_directory`() methods.

In addition, a neuray installation can be extended with importer and exporter plugins that support other file formats. Such plugins are written using the Iray API. They need to be loaded explicitly in neuray using the `IPlugin_configuration::load_plugin_library`() or `IPlugin_configuration::load_plugins_from_directory`() methods. The latter allows you to offer automatic plugin loading from a specified directory. New importers are derived from the IImporter interface and new exporters are derived from the IExporter interface.

The following interfaces and function calls work with scene data and file formats:

- `IImport_api`
- `IExport_api`

18.1.3 The .axf material file format

The AxF file format from X-Rite stores digital material representations, most notably measured materials from their BTF scanning technology. The format supports different representations categorized in profiles. neuray supports the profiles for spatially varying BRDF representations with non-refracting clearcoat (profile "AxFSvbrdf," versions 1, 2 and 3), for car paint representations (profiles "AxFCarPaint" and AxFCarPaintRefract," version 1) and for volumetric representations (profile "AxFVolumetric," version 1). These representations use a simple material model and a set of textures to control the input parameters of those material models.

The AxF file format is supported in neuray with an import plugin for the `.axf` file extension and a corresponding MDL module. Once both parts have been set up properly, an import of

AxF files works in the same way as an import of MDL modules. The result of an AxF file import is then a MDL material in the neuray database with the respective textures from the AxF file set for its parameters. If available, additional preview images are also imported into the database.

18.1.3.1 Prerequisites before importing AxF files

1. The AxF importer is a neuray plugin, which is contained in the neuray release. To enable AxF file support in your application you need to load the plugin explicitly in neuray using the `IPlugin_configuration::load_plugin_library`() or `IPlugin_configuration::load_plugins_from_directory`() methods. The latter allows you to offer automatic plugin loading from a specified directory.
2. For the importer to function properly, you need to place the MDL file `nvidia/axf_importer/axf_importer.mdl`, which is in the `mdl` directory of the neuray release, with its directory structure in a place where neuray will find it in your application. In particular, your MDL search path needs to be configured to find the `nvidia` directory. See `IRendering_configuration` for more details on how to configure the search path.
3. AxF files are imported in the same way as MDL files, thus the MDL search path must be configured by you to point to the AxF files. See `IRendering_configuration` for more details on how to configure the search path.

18.1.3.2 Importing AxF files

Axf files with the `.axf` extension are imported in the same way as MDL files. You can import them using the corresponding API calls and you can include them in .mi files.

- You can import AxF files with the neuray API function `IImport_api::import_elements`(). Similar to MDL file imports, URIs to AxF files cannot be given as absolute paths and the directory of the AxF file has to be in the configured set of MDL paths, as explained in the previous section. A possible URI could be:

  ```
  "{shader}/axf_file.axf"
  ```

- You can use the regular `include` statement in a .mi file to import an AxF file, for example:

  ```
  $include "axf_file.axf"
  ```

18.1.3.3 Import results in the database

The AxF importer loads the AxF file and converts its base profile representation to an equivalent MDL material. It creates the following elements in the neuray database:

1. The textures for the base profile representation.
2. A new MDL material is created as a variant of the `nvidia/axf_importer/axf_importer/svbrdf` or `nvidia/axf_importer/axf_importer/carpaint` material, where the imported textures are used as new default values for the respective `texture2d` parameters. Those parameters are annotated as `hidden()` parameters, while the other parameters remain

visible for later parameter editing when the material is instantiated and assigned to objects.

For example, if the AxF file is called `example_file.axf`, then the corresponding MDL material will be located in a new MDL module named `mdl::axf::example_file`, and the MDL material will be named like the AxF material in the AxF file, for example `Leather`, followed by the representation (currently always `_svbrdf` or `_carpaint`) The fully qualified MDL material name is then in this example `mdl::axf::example_material::Leather_svbrdf`.

The new MDL module is not automatically saved to disk.

3. AxF files can contain preview images of a material at different resolutions. They are also imported and placed in the database.

If you use the API function `IImport_api::import_elements`(), you can get the database names of all imported elements from the returned import result.

18.1.4 The mi_importer plugin

Use the mi_importer plugin to enable the import of a scene to the NVIDIA Iray .mi file format. To enable the plugin it must be loaded in neuray using either the `IPlugin_configuration::load_plugin_library`() or the `IPlugin_configuration::load_plugins_from_directory`() method.

18.1.4.1 Importer configuration

In addition to the standard options described for `IImporter::import_elements`, the mi_importer plugin has an additional configuration option:

- `mi_progress_callback` of type `IProgress_callback`. When configured, this callback is used to report the progress of the import operation.

18.1.5 The mi_exporter plugin

Use the mi_exporter plugin to enable the export of a scene to the NVIDIA Iray .mi file format. To enable the plugin it must be loaded in neuray using either the `IPlugin_configuration::load_plugin_library`() or the `IPlugin_configuration::load_plugins_from_directory`() method.

18.1.5.1 Exporter configuration

In addition to the standard options described for `IExporter::export_elements` and `IExporter::export_scene`, the mi_exporter plugin has additional configuration options:

- `mi_write_binary_vectors_limit` of type `mi::IUint32`. Geometric objects with vectors larger than this size are exported in a binary format. It saves space and speeds up both export and import, while making the data humanly unreadable.
- `mi_mdl_new_naming_scheme` of type `mi::IBoolean`. Determines whether to use the new MDL naming scheme for function calls and material instances. Default: `true`.
- `mi_mdl_root` of type `mi::IString`. When exporting memory-based MDL modules, the given value defines the location of the root directory to which the MDL data is exported.

- `mi_mdle_export_mode` of type `mi::IString`: Defines, how MDLE files are handled during the export. The following modes are supported:
  - `absolute` MDLE files are exported using absolute paths. This is the default behavior.
  - `relative` MDLE file paths are made relative to the scene export location, if possible.
  - `bundle` MDLE files are copied next to the scene or into the directory defined with the exporter option `mi_mdle_export_directory`.
- `mi_mdle_export_directory` of type `mi::IString`: the name/path of the directory MDLE files should be copied to in `bundle` - mode.
- `mi_uvtile_mode_marker` of type `mi::IString`: The preferred export-mode for memory based UDIM textures. Can be any of `<UVTILE0>`, `<UVTILE1>`, `<UDIM>`. Default: `<UVTILE0>`.
- `mi_create_verbatim_textures` of type `mi::IBoolean`: Determines whether verbatim (inline) textures will be created (if necessary and possible), or whether such textures will be exported into separate files instead. Default: `true`. Default: `<UVTILE0>`.

18.1.6 Image file formats

neuray has no built-in support for reading and writing image data in different file formats. This support is solely provided by image plugins that can be dynamically loaded.

The following image plugins are currently available:

`nv_openimageio`
: The OpenImageIO library[1] offers support for various image formats, including `EXR`, `GIF`, `HDR`, `JPG`, `PNG`, and `TIFF`.

`dds`
: The DDS (DirectDraw Surface) image format was established by Microsoft. The use of this plugin is recommended to add support for 3D textures, cubemaps, and mipmaps.

`ct`
: The CT format is a lossless image format created by mental images.

Your neuray installation can be extended with image plugins that support other file formats.

Every image plugin needs to be loaded explicitly in neuray using the `IPlugin_configuration::load_plugin_library`() or `IPlugin_configuration::load_plugins_from_directory`() methods. The latter allows you to offer automatic plugin loading from a specified directory.

18.1.7 Binary storage format

neuray can read a special binary storage format which is optimized for fast reading and needs less disk space than a typical `.mi` file. Files containing this format are required to have the `.mib` file name extension. Files in the `.mib` format can be generated from any of the supported input file formats using the export functionality of neuray; see the `IExport_api::export_scene`() and `IExport_api::export_elements`() functions. The `.mib`

1. `https://github.com/OpenImageIO/oiio`

file contains basically the same information as the file from which it was generated only in an optimized format.

Note: Be sure to use the same version of neuray when exporting (writing) and reading a particular `.mib` file. If you use a different version of neuray to read the `.mib` file, neuray will not read the file and an error message will be displayed.

18.2 Render target canvases

The canvas types reflected by the `IRender_target_base::get_canvas_type()` function determine what is rendered to the canvas. Each type of canvas has a native pixel type that best encodes the data generated by the renderer. This type is listed below with the description of each canvas type. Note that applications may choose to use a different type. Pixels will be converted accordingly.

Please note that not all canvas types are supported everywhere. See the limitations subsection in the individual render mode sections for the relevant limitations on canvases.

A list of supported canvas types and their optional parameters follows. Most canvas types support an optional parameter `PARAM_PROCESSING_DISABLED`, which defaults to false. Setting this parameter to true on a canvas will disable internal postprocessing of that canvas and instead write the raw output of the renderer to it.

`TYPE_RESULT`

An `mi::math::Color` or `mi::Float32_3` canvas that contains the rendered image with all contributions, the *beauty pass*, including the alpha channel if the pixel type supports it.

The behavior of the result buffer may be controlled with the following, optional parameters:

Canvas_parameter	*Type*	*Default*	*Effect*
`PARAM_COLOR_LPE`	`mi::IString`	`nullptr`	Controls the LPE for the color channels. If provided, contributions to the canvas are limited to paths which match the given expression.
`PARAM_ALPHA_LPE`	`mi::IString`	`nullptr`	Controls the LPE for the alpha channel, if available in the canvas. The default that is used if no value is provided is controlled by the `"iray_default_alpha_lpe"` option. Note that a primary feature of matte objects is to catch shadows. Shadows cast by synthetic objects upon matte objects may make the alpha channel opaque. Whether or not this is the case depends on the active alpha LPE.

`PARAM_SCALE`	`mi::Float32`	1	Controls upscaling. A value of 1 indicates regular rendering, while a value of 2 will cause the rendered image to be scaled to double its original resolution in width and height. Upscaling is handled by the Deep Learning-based Denoiser (page 38), which must be made available for this option to be accepted. While the denoiser must be available, it does not need to be globally active, i.e. upscaling implies denoising for upscaled canvases without mandating denoising of other canvases. The canvas resolution must meet the requirements described in the subwindow group of the `ICamera` interface, where all values are multiplied by the upscaling factor. Note that this is true regardless of whether the denoiser is actually active.

Light path expressions are explained in depth in a separate section (page 190). Some common examples are:

Expression	*Resulting image*
`E D .* L`	Diffuse pass commonly used in conventional compositing workflows. The last event on the light path before the eye was a diffuse event.
`E G .* L`	Glossy pass commonly used in conventional compositing workflows. The last event on the light path before the eye was a glossy event.
`E S .* L`	Specular pass commonly used in conventional compositing workflows. The last event on the light path before the eye was a specular event, that is, a mirror reflection or specular refraction.
`E D S .* L`	Diffuse part of caustics cast by a mirror reflection or specular refraction on another surface.
`E .* <L'key'>`	All direct and indirect light contribution coming from the `key` light group, which are all lights in the scene with the `handle` attribute set to `key`.
`E 'crate' .* L`	All direct and indirect light falling onto any object in the scene with the `handle` attribute set to `crate`.

The alpha channel of each sample may be transparent (alpha 0) only if the alpha LPE matches the light transport path. Common examples of LPEs for the alpha channel are:

Expression	*Resulting image*
`E [LmLmsLe]`	Alpha is based solely on primary visibility. This is the approach used traditionally by many renderers.
`E [LmLe]`	Alpha is based on primary visibility. Matte shadows are opaque.
`E T* [LmLe]`	Transmitting objects make the alpha channel transparent. This is the default behavior.
`E <TS>* [LmLe]`	Only specular transmission makes the alpha channel transparent. This avoids unexpected results in scenes with materials that have a diffuse transmission component.

The API provides a utility function, `IRendering_configuration::make_alpha_expression`(), to generate useful alpha expressions including the ones listed above, and `IRendering_configuration::make_alpha_mask_expression`() to generate alpha mask expressions.

Note that alpha LPEs operate independently of color LPEs. In particular, the paths that affect the alpha channel of a RGBA canvas are not restricted to those that affect the color channels.

See "LPEs for alpha control" (page 194) for more information on alpha LPEs.

`TYPE_ALPHA`

An `mi::Float32` canvas that contains the alpha channel. This canvas works in analogy to the `TYPE_RESULT` canvas described above, including the optional use of `PARAM_ALPHA_LPE`. Other parameters are not supported.

`TYPE_BSDF_WEIGHT`

An `mi::Float32_3` canvas that contains approximate color weights for the constituent BSDFs of the material at the first hit. This value will generally reflect textures (bitmap and procedural) used to color material components. Note that the exact values are subject to change.

`TYPE_DEPTH`

An `mi::Float32` canvas that contains the depth of the hit point along the (negative) z-coordinate in camera space. The depth is zero at the camera position and extends positive into the scene.

`TYPE_RAYLENGTH`

An `mi::Float32` canvas that contains the distance between the camera and the hit point measured in camera space. The depth is zero at the camera position and extends positive into the scene. The distance is related to `TYPE_DEPTH` by a factor that is the inner product of the (normalized) ray direction and the negative z-axis (0,0,-1).

`TYPE_NORMAL`

An `mi::Float32_3` canvas that contains the normalized surface shading normal in camera space including the averaged effect from normal perturbation functions.

`TYPE_WORLD_POSITION`
An `mi::Float32_3` canvas that contains the position of the first hit in world space.

`TYPE_TEXTURE_COORDINATE`
An `mi::Float32_3` canvas that contains the *i-th* texture coordinate at the hit point, where *i* is stored in the canvas parameter `PARAM_INDEX`.

`TYPE_OBJECT_ID`
An `mi::math::Color` canvas that contains the ID for the scene element at the hit point. Each scene element can have an optional attribute named `label` of type `mi::Uint32` which defines this ID. IDs are mapped to colors by applying the bytes of the ID in order of increasing significance to 8bit RGBA. If this scene element is imported from a `.mi` file, this label can be set using the `tag` statement in `.mi` file syntax.

`TYPE_MATERIAL_ID`
An `mi::math::Color` canvas that contains the user-defined material ID of the material at the hit point. This ID can be used to identify regions in the image that show the same material or regions without material. The latter are colored black. IDs are mapped to colors by applying the bytes of the ID in order of increasing significance to 8bit RGBA.

`TYPE_MATERIAL_TAG`
An `mi::math::Color` canvas that contains an internal implementation-specific ID of the surface material at the hit point. This ID can be used to identify regions in the image that show the same material or regions without material. The latter are colored black. IDs are mapped to colors in an implementation-specific manner.

`TYPE_MULTI_MATTE`
A scalar or vector canvas that contains one mask per channel. One to four channels can be combined in a single canvas and multiple canvases of this type can be rendered in parallel.

Each channel contains a coverage mask that is essentially equivalent to the result of rendering an alpha channel with the LPE provided by `IRendering_configuration::make_alpha_mask_expression`(handle,false,ALPHA_PRIMARY) , except that the selection is based on object IDs (the instance's `label` attribute) or material IDs (the material's `material_id` attribute) instead of a handle string.

The canvas parameter `PARAM_MULTI_INDEX` controls the generation of the masks. Each of the vector's components controls the contribution to the respective pixel component. Positive values will yield a mask containing visible elements where the object ID matches the provided value. Negative values will yield a mask containing visible elements where the material's ID matches the provided value after flipping its sign. A component value of 0 is interpreted as a terminator. Any subsequent non-zero components are ignored.

Note that the number of requested masks must not be larger than the number of channels in the canvas' pixel type.

While this canvas reproduces behavior found in a few other rendering packages, users should note that alpha LPEs are generally more flexible than simple visibility/coverage masks. More importantly, compositing workflows based on color LPEs will generally yield superior results than workflows based on ID or alpha masks.

`TYPE_SHADOW`
An `mi::math::Color` canvas that contains the shadow in the scene. More precisely, the canvas contains the light contributions that are missing at a certain point because it is

blocked by a some object, the shadow casters. With this definition, adding the `shadow` canvas to the `result` canvas removes the shadow. In practice, a weighted version of the `shadow` buffer can be added; with a small positive weight to lighten the shadow or a small negative weight to darken the shadow.

`TYPE_AMBIENT_OCCLUSION`

An `mi::math::Color` canvas that contains the ambient occlusion in the scene in the range from 0 (fully occluded) to 1 (not occluded). The computation of the buffer can be controlled using the following attributes on the `IAttribute_set` class:

`mi::Float32` ambient_falloff_max_distance = FLT_MAX

World space distance beyond which potential occlusion does not influence result of ambient occlusion.

`mi::Float32` ambient_falloff_min_distance = FLT_MAX

World space distance below which all potential occlusion fully influences result of ambient occlusion.

`mi::Float32` ambient_falloff = 1

Exponent to be used for weighting the occlusion value between falloff min and max distance. The default 1 blends linearly.

Note that the Iray Interactive render mode offers additional options (page 67) that affect the ambient occlusion buffer.

`TYPE_RESULT_IRRADIANCE`

An `mi::Float32_3` canvas that contains the estimated irradiance at the hit point. The canvas contains black for direct environment hits.

Like the `TYPE_RESULT` canvas, the contents of this canvas can be restricted to the light transport paths that match the expression obtained by querying the canvas parameters for `PARAM_COLOR_LPE`. Consider using irradiance expressions for this canvas.

Also make sure that in the rendering options (page 56) `"iray_nominal_luminance"` and `"iray_firefly_filter"` are properly setup to avoid wrong simulation behavior.

In addition, make sure that no tonemapping is being used on the result, as it will remap the simulated irradiance values, thus rendering them useless for lighting measurements and verification.

`TYPE_RESULT_IRRADIANCE_PROBE`

An `mi::Float32_3` canvas that contains the estimated irradiance at the probe points defined in the `IIrradiance_probes` container attached to the `ICamera` .

Like the `TYPE_RESULT` canvas, the contents of this canvas can be restricted to the light transport paths that match the expression obtained by querying the canvas parameters for `PARAM_COLOR_LPE`. Consider using irradiance expressions for this canvas.

Also make sure that in the rendering options (page 56) `"iray_nominal_luminance"` and `"iray_firefly_filter"` are properly set up to avoid wrong simulation behavior.

In addition, make sure that no tonemapping is being used on the result, as it will remap the simulated irradiance values, thus rendering them useless for lighting measurements and verification.

`TYPE_MOTION_VECTOR`

An `mi::Float32_3` canvas that contains the motion vectors.

`TYPE_CONVERGENCE_HEATMAP`

Generates a `mi::Float32` convergence heatmap for the render target canvas *i*, where *i* is obtained by querying `PARAM_INDEX`. The referenced canvas must be of type `TYPE_RESULT` or `TYPE_SHADOW`. Note that the referenced canvas must precede this canvas in the render target, i.e., forward references are not allowed.

If the SSIM predictor is not enabled, this canvas is not supported.

`TYPE_POST_TOON`

An `mi::math::Color` or `mi::Float32_3` canvas. The contents are generated by applying a cartoon shading effect to the *i-th* render target canvas, where *i* is obtained by querying `PARAM_INDEX`. This effect may be applied to canvases of type `TYPE_RESULT`, `TYPE_AMBIENT_OCCLUSION`, and `TYPE_BSDF_WEIGHT`, provided that those canvases are not upscaled. Note that the referenced canvas must precede this canvas in the render target, i.e., forward references are not allowed.

The cartoon shading effect depends on object IDs for edge detection. Users should ensure that IDs are assigned in a sufficiently random fashion to support edge detection.

It is recommended to enable the "progressive_aux_canvas" scene option and to disable any camera/lens effects such as depth of field (unless the sample count is high enough).

For canvases other than `TYPE_RESULT`, the background will be replaced with the scene environment or backplate, as applicable. This background will pass through the same postprocessing operations (e.g. denoising and tonemapping) as any canvas of type `TYPE_RESULT`. Note that this is true even if no result canvas is otherwise present in the render target.

The details of the cartoon effect are controlled by the following canvas parameters.

Canvas_parameter	*Type*	*Default*	*Effect*
`PARAM_INDEX`	`mi::Sint32`	0	Selects the canvas which serves as the input to the cartoon effect, as described above.
`PARAM_EDGE_COLOR`	`mi::Float32_4`	(0,0,0,1)	Controls the color of object outlines.
`PARAM_SCALE`	`mi::Float32`	0	Controls a faux surface shading effect if the toon input canvas is not `TYPE_RESULT`. Values < 0 enable smooth faux lighting based on surface BSDF weight and normal orientation. A value of 0 uses the toon input canvas as-is. Values > 0 apply a quantization effect to the faux lighting for a cartoon look.

An object ID of 0 may be used to exclude objects from receiving cartoon edges. Such objects will still receive the optional faux lighting effects where applicable.

It is also possible to use the toon effect without any input canvas by supplying a `PARAM_INDEX` of -1. In this case, edges (and optionally faux lighting effects) will be applied to a white canvas. No background replacement will take place.

`TYPE_SELECTION_OUTLINE`

An `mi::math::Color` or `mi::Float32_3` canvas. The contents are generated by drawing an outline around all scene elements which have a boolean attribute "selected" set to `true`.

The effect is overlaid onto the *i-th* render target canvas, where *i* is obtained by querying `PARAM_INDEX`. This effect may be applied to any canvas that is not upscaled. Note that the referenced canvas must precede this canvas in the render target, i.e., forward references are not allowed. It is also possible to use the outline effect without any input canvas by supplying a `PARAM_INDEX` of -1. In this case, edges will be applied to a black, transparent canvas.

The outline effect partially depends on object IDs for edge detection. Users should ensure that IDs are assigned in a sufficiently random fashion to support edge detection.

It is recommended to disable the "progressive_aux_canvas" scene option and any camera/lens effects such as depth of field.

The details of the outline effect are controlled by the following canvas parameters.

Canvas_parameter	*Type*	*Default*	*Effect*
`PARAM_INDEX`	`mi::Sint32`	0	Selects the canvas onto which outlines are written, as described above.
`PARAM_SCALE`	`mi::Float32`	2	Controls the width of object outlines.
`PARAM_EDGE_COLOR`	`mi::Float32_4`	(0,0,0,1)	Controls the color of object outlines between selected and unselected objects or the environment.
`PARAM_EDGE_COLOR_2`	`mi::Float32_4`	(0,0,0,0)	Controls the color of object outlines between selected objects.
`PARAM_SHADE_COLOR`	`mi::Float32_4`	(0,0,0,0)	Controls a color overlaid over the entire selected object.

18.2.1 Canvas names

Previous releases of this API encoded canvas type and parameter in a single string. A mapping between these render target canvas names and canvas types and parameters is listed below. The same mapping is also available via the API of `IRendering_configuration::map_canvas_name`() and `IRendering_configuration::map_canvas_parameters`(). Note, however, that the combination of canvas type and parameters is more powerful and not all information will be retained in the generated string.

`result`, `result lpexpr=` *lp-expressions*, or `lpexpr=` *lp-expressions*
: Corresponds to `TYPE_RESULT`.

  The optional `lpexpr=...` part specifies either one light path expression, or two LPEs separated by a semicolon. These LPEs correspond to canvas parameters of `PARAM_COLOR_LPE` and `PARAM_ALPHA_LPE`, respectively.

`alpha` or `alpha lpexpr=` *lp-expression*
: Corresponds to `TYPE_ALPHA` with an optional `PARAM_ALPHA_LPE`.

`depth`
: Corresponds to `TYPE_DEPTH`.

`distance`
: Corresponds to `TYPE_RAYLENGTH`.

`normal`
Corresponds to `TYPE_NORMAL`.

`texture_coordinate[` *i* `]`
Corresponds to `TYPE_TEXTURE_COORDINATE` with *i* mapped to a canvas parameter of type `PARAM_INDEX`.

Omitting the index expression, i.e. a name of "texture_coordinate", is allowed and is equivalent to "texture_coordinate[0]".

`object_id`
Corresponds to `TYPE_OBJECT_ID`.

`material_id`
Corresponds to `TYPE_MATERIAL_ID`.

`generated_material_id`
Corresponds to `TYPE_MATERIAL_TAG`.

`diffuse`
A canvas that contains all light transport paths which are diffuse at the first bounce. This corresponds to `TYPE_RESULT` with a `PARAM_COLOR_LPE` of `"E D .* L"`.

`specular`
A canvas that contains all light transport paths which are specular at the first bounce. This corresponds to `TYPE_RESULT` with a `PARAM_COLOR_LPE` of `"E S .* L"`.

`glossy`
A canvas that contains all light transport paths which are glossy at the first bounce. This corresponds to `TYPE_RESULT` with a `PARAM_COLOR_LPE` of `"E G .* L"`.

`emission`
A canvas that contains the emission contribution from directly visible light sources and emitting surfaces. This corresponds to `TYPE_RESULT` with a `PARAM_COLOR_LPE` of `"E L"`.

`shadow`
Corresponds to `TYPE_SHADOW`.

`ambient_occlusion`
Corresponds to `TYPE_AMBIENT_OCCLUSION`.

`irradiance` or `irradiance lpexpr=` *lp-expression*
Corresponds to `TYPE_RESULT_IRRADIANCE` with an optional LPE.

Like the `result` canvas, the contents of the `irradiance` canvas can be restricted to the light transport paths that match *lp-expression*. Consider using irradiance expressions for this canvas.

`irradiance_probe` or `irradiance_probe lpexpr=` *lp-expression*
Corresponds to `TYPE_RESULT_IRRADIANCE_PROBE` with an optional LPE.

Like the `result` canvas, the contents of the `irradiance_probe` canvas can be restricted to the light transport paths that match *lp-expression*. Consider using irradiance expressions for this canvas.

`motion_vector`
Corresponds to `TYPE_MOTION_VECTOR`.

`convergence_heatmap[` *i* `]`
Corresponds to `TYPE_CONVERGENCE_HEATMAP` for the result buffer *i*. Note that that buffer must precede this buffer in the render target, i.e., forward references are not allowed.

Omitting the index expression, i.e. a name of "convergence_heatmap", is allowed and is equivalent to "convergence_heatmap[0]".

`toon[` *i* `]`
Corresponds to `TYPE_POST_TOON` for the buffer *i*. Note that the referenced canvas must precede this canvas in the render target, i.e., forward references are not allowed. Also note that no other parameters may be encoded in this canvas name. Please consider using `TYPE_POST_TOON` and `ICanvas_parameters`.

Omitting the index expression, i.e. a name of "toon", is allowed and is equivalent to "toon[0]".

`selection_outline[` *i* `]`
Corresponds to `TYPE_SELECTION_OUTLINE` for the buffer *i*. Note that the referenced canvas must precede this canvas in the render target, i.e., forward references are not allowed. Also note that no other parameters may be encoded in this canvas name. Please consider using `TYPE_SELECTION_OUTLINE` and `ICanvas_parameters`.

Omitting the index expression, i.e. a name of "selection_outline", is allowed and is equivalent to "selection_outline[0]".

`multi_matte[` *id [, ...]* `]`
Corresponds to `TYPE_MULTI_MATTE`. Between one and four IDs control the IDs used to generate the masks.

18.3 Light path expressions

18.3.1 Introduction

Light path expressions (LPEs) describe the propagation of light through a scene, for example starting from a source of light, bouncing around between the objects of the scene and ultimately ending up at the eye. The paths that light takes through a scene are called *light transport paths*. LPEs may be used for example, to extract only specific light contributions from a renderer into separate image buffers.

Light path expressions were first proposed as a description of light transport by Paul Heckbert [Heckbert90] (page 197). Heckbert suggested that *regular expressions* — typically used to describe patterns of characters in text — could also describe light transport events and paths. The alphabet of LPEs consists of event descriptions, that is, of interactions between light particles and the scene.

18.3.2 Events

Each event of a path, that is, each interaction of light with the scene objects and materials, is described by its type (for example, emission or reflection), the mode of scattering (for example, diffuse or specular), and an optional handle.

A full event is described as `< t m h >`, where

- `t` is the event type, either `R` (reflection), `T` (transmission), or `V` (volume interaction),
- `m` is the scattering mode, either `D` (diffuse), `G` (glossy), or `S` (specular), and
- `h` is a handle in single quotes, for example. `'foo'`. This position may be omitted from the specification. In that case, any handle is accepted. See below for details.

Spaces are ignored unless they occur inside a handle string. The dot character (`.`) may be used as a wildcard in any position. It accepts any valid input. For example, a diffuse reflection event may be specified as `<RD.>`, or, omitting the handle, `<RD>`. A specular transmission event identified with the handle `"window"` may be specified as `<TS'window'>`.

18.3.3 Handles

Handles are strings of ASCII characters, enclosed in single quotes (`'`). The following characters must be escaped by prefixing them with a backslash inside handles: `\`, `'`, and `"`. The assignment of a handle as a name for a scene element is typically made possible through the graphical interface of an application.

18.3.4 Sets and exclusion

As an alternative to the type, mode, and handle specifiers described above, each position of the event triple may contain a character set enclosed in square brackets. Any element of the set will be accepted. For example, `<[RT]..>` matches all reflection events and all transmission events.

The complementary set is specified by first including the caret (`^`) character. For example, `<.[^S]>` matches any non-specular event and `<..[^'ground']>` matches any event that is not identified with the handle "ground".

Event sets also work on full events. For instance, `[<RG><TS>]` matches glossy reflection and specular transmission events. Note that this is different from `<[RT][GS]>`, which accepts glossy transmission and specular reflection in addition to the events accepted by the previous expression.

18.3.5 Abbreviations

In order to make specification of LPEs simpler, event descriptions may be abbreviated. An event in which only one of type, mode, or handle is explicitly specified may be replaced by that item. For example, `<R..>` may be abbreviated as `R`. Likewise, `<..'foo'>` may be abbreviated as `'foo'`. Note the difference between `<TS.>`, which accepts a single specular transmission event, and `TS`, which accepts an arbitrary transmission event followed by an arbitrary specular event. Finally, `.` matches any event except the special events described below.

Abbreviation rules also apply to event sets, that is, `[<T..><.S.>]` reduces to `[TS]`. Again note that this is different from `TS` (without brackets).

18.3.6 Constructing expressions

LPEs may be constructed by combining event specifications through concatenation, alternation, and quantification. The following operators are supported and have the same

semantics as they would for standard regular expressions. For expressions `A` and `B` and integers `n` and `m` with `m >= n`:

`AB`	Accepts first `A`, then `B`
`A|B`	Accepts `A` or `B`
`A?`	Optionally accepts `A`, that is, `A` may or may not be present
`A*`	Accepts any number of occurrences of `A` in sequence, including zero times
`A+`	Accepts any non-empty sequence of `A`'s. It is equivalent to `AA*`
`A{n}`	Accepts exactly `n` consecutive occurrences of `A`. For example, `A{3}` is equivalent to `AAA`.
`A{n,m}`	Accepts from `n` to `m`, inclusively, occurrences of `A`
`A{n,}`	Equivalent to `A{n}A*`

The precedence from high to low is quantifiers (`?`, `*`, `+`, `{}`), concatenation, alternatives. Items can be grouped using normal parentheses `(` and `)`.

18.3.7 Special events

Each LPE is delimited by special events for the eye (or camera) and light vertices. These events serve as markers and must be the first and last symbols in a LPE.

LPEs may be specified starting either at the light or at the eye. All expressions must be constructed in such a way that every possible match has exactly one eye and one light vertex. This is due to the nature of LPEs: They describe light transport paths between one light and the eye. Note that this does not mean that light and eye markers must each show up exactly once. For example, `"E (D La | G Le)"` is correct, because either alternative has exactly one light and one eye marker: `"E D La"` and `"E G Le"`. On the other hand `"E D La?"`, `"E (D | La)"`, and `"E (D | La) Le"` are ill-formed, because they would match paths with zero or two light markers.

In the abbreviated form, the eye marker is simply `E` and the light marker is `L`. These items are special in that they represent two distinct characteristics: the shape of the light (or camera lens) and the mode of the emission distribution function. The full notation therefore differs from that of the standard events.

In the full form, a light source as the first vertex of a transport path is described as `< L h m h >`, where `L` is the light type, and `m` and `h` are as before. The first pair of type and handle describes the light source itself. The type `L` can be one of `Lp` (point shape), `La` (area light), `Le` (environment or background), `Lm` (matte lookup), or `L` (any type). In the case of alpha expressions, `Lms` (matte shadows) is supported in addition to the aforementioned types. The second pair describes the light's directional characteristics, that is, its EDF. (This form loosely corresponds to the full-path notation introduced by Eric Veach in [Veach97] (page 197), Section 8.3.2.)

As before, the handles are optional. Furthermore, the EDF specification may be omitted. Thus, `<La>` is equivalent to `<La...>` and `La`.

Especially when dealing with irradiance (rather than radiance) render targets, it is convenient to use a special form of LPE, called *irradiance expression*. Such expressions contain an

irradiance marker, rather than an eye marker. Using this marker, it is possible to describe light transport up to the point of the irradiance measurement, rather than up to the camera.

The full form of the irradiance marker is `<I h>`, the abbreviated form is simply `I`. As before, `h` represents an optional handle. If set, irradiance will only be computed on those surfaces that have a matching handle.

18.3.8 Advanced operations

Several operations exist in order to make specifying the right expression easier. These operations do not add expressive power, but simplify certain tasks.

When an application provides a means for multiple output images (or *canvases*) to be rendered at the same time, expressions may be re-used in subsequent canvas names. This is achieved by assigning a name to the expressions that should be re-used, for example:

1. `caustics: L.*SDE`
2. `LE | $caustics`

In this example, the second canvas will receive both caustics and directly visible light sources. As illustrated above, variables are introduced by specifying the desired name, followed by a colon and the desired expression. Variable names may contain any positive number of alphanumeric characters and the underscore character, with the limitation that they may not start with a terminal symbol. Since all terminals of the LPE grammar start with capital letters, it is good practice to start variable names with lowercase letters. Note that sub-expressions cannot be captured by variable names.

Variables are referenced by applying the dollar (or value-of) operator to the variable name.

Expressions may be turned into their complement by prefixing them with the caret symbol. An expression of type `^A` will yield all light transport paths that are not matched by `A`. Note that the complement operator cannot be applied to sub-expressions: `"^(L.*E)"` is valid, but `"L^(.*)E"` is not.

It is possible to compute the intersection of two or more expressions by combining them with the ampersand symbol. Expressions of type `A & B` will match only those paths which are matched by both `A` and `B`.

18.3.9 Matte object interaction

The color of matte objects is determined by two types of interaction. The first is the lookup into the environment or backplate. This contribution is potentially shadowed by other objects in the scene and may be selected by expressions ending in `Lm`. Selection of such contributions can be further refined by specifying the handle of the matte object, for example, `<Lm 'crate'>`.

The second type of contributions is made up of effects that synthetic objects and lights have on matte objects. This includes effects like synthetic lights illuminating matte objects and synthetic objects reflected by matte objects. For these contributions, matte objects behave exactly like synthetic objects.

With regards to LPEs, matte lights illuminating synthetic objects behave exactly as if they were synthetic lights.

18.3.10 LPEs for alpha control

Some additional considerations are necessary when using LPEs to control alpha output.

By definition, alpha is transparent (alpha 0) only for paths that match the provided expression. Note that this means that light transport paths which do not reach a light source or the environment because they are terminated prematurely (for whatever reason) are opaque. This is necessary to avoid undesired semi-transparent areas in the standard case.

This has implications for the creation of object masks. Since the mask is supposed to be completely transparent also when undesired objects are hit by camera rays, these paths have to be captured by the expression even if they are terminated. This requires a special type of LPE, that is, one that captures terminated paths. Such LPEs are only allowed for alpha channels. For example, the expression `"E ([^'crate'] .*)? L?"` will render a mask for the object `'crate'`.

Shadows received by matte objects may also affect the opacity of the alpha channel. This opacity can be removed by capturing paths which end in `Lms`.[2] Adding opacity in areas of matte shadow to the previously shown mask expression may be achieved by slightly changing the LPE to `"E ([^'crate'] .*) L? | E [^Lms]"`.

Note that presence or absence of `Lms` controls whether shadows which are received by a certain matte object make the alpha channel opaque. This affects all matte shadow, regardless of how it was cast, and by which objects.

The API provides functions `IRendering_configuration::make_alpha_expression`() and `IRendering_configuration::make_alpha_mask_expression`() to generate various common alpha expressions, including masks.

18.3.11 Example LPEs

The universal light path expression, `"L .* E"`, accepts all light transport paths. By default, this expression yields the same result as not using LPEs at all. Remember that this is equivalent to `"L.*E"` (whitespace is ignored) and `"E.*L"` (the expression can be reversed).

Direct illumination can be requested by specifying `"L .? E"`, or `"L . E"` if directly visible light sources are not desired. Indirect illumination is then specified by `"L .{2,} E"`.

Compositing workflows often use the concept of diffuse and reflection passes. They can be specified with the LPEs `"E <RD> L"` and `"E <RS> L"`, respectively. Note that these passes as specified above do not contain indirect illumination. `"E <RD> .* L"` extends this to global illumination where the visible surfaces are diffuse. If only diffuse interactions are desired, `"E <RD>* L"` can be used.

Caustics are usually described as `"E D S .* L"`. The expression `"E D (S|G) .* L"` or `"E D [GS] .* L"` also considers glossy objects as caustics casters. If, for example, only specular reflection caustics cast by an object identified with a handle "crate" are desired, the expression is changed to `"E D <RS'crate'> .* L"`. This can further be restricted to caustics cast onto "ground" from a spot light by changing the expression to `"E 'ground' <RS'crate'> .* <LpG>"`.

2. The mask expression in the previous paragraph captures all light types and thus also makes matte shadows transparent.

Assuming an expression variable called `caustics` was defined in a previous expression, `"L.{2,5}E & ^$caustics"` will match any path that has the specified length and is not a caustic.

18.3.12 Summary of typical LPEs

Typical production workflow structures in digital compositing often employ a set of standard elements that can be represented by light path expressions. The following expressions define the color (RGB) component of rendering:

LPE	*Description*
`E D .* L`	Diffuse pass commonly used in conventional compositing workflows. The last event on the light path before the eye was a diffuse event.
`E G .* L`	Glossy pass commonly used in conventional compositing workflows. The last event on the light path before the eye was a glossy event.
`E S .* L`	Specular pass commonly used in conventional compositing workflows. The last event on the light path before the eye was a specular event, that is, a mirror reflection or specular refraction.
`E D S .* L`	Diffuse part of caustics cast by a mirror reflection or specular refraction on another surface.
`E .* <L'key'>`	All direct and indirect light contribution coming from the `key` light group, which are all lights in the scene with the `handle` attribute set to `key`.
`E 'crate' .* L`	All direct and indirect light falling onto any object in the scene with the `handle` attribute set to `crate`.

The alpha channel can also be specified by light path expressions:

LPE	*Description*
`E [LmLe]`	Alpha is based solely on primary visibility. This is the approach used traditionally by many renderers.
`E T* [LmLe]`	Transmitting objects make the alpha channel transparent. This is the default behavior.
`E <TS>* [LmLe]`	Only specular transmission makes the alpha channel transparent. This avoids unexpected results in scenes with materials that have a diffuse transmission component.

The graphical interface of an application may provide a way of naming and storing LPEs for reuse. Common LPEs like the above may also be part of a standard set of named LPEs in an application interface.

18.3.13 Light path expression grammar

`L`	light
`E`	eye
`R`	reflection type
`T`	transmission type
`V`	volume interaction type
`D`	diffuse mode
`G`	glossy mode
`S`	specular mode
`'`*h*`'`	handle *h*
`<` *type mode handle* `>`	event
`Lp`	point light type
`La`	area light type
`Le`	environment or background light type
`Lm`	matte lookup type
`Lms`	shadows cast onto matte objects (alpha expressions only)
`<` *light-type light-handle mode handle* `>`	light source full form
`<I` *h* `>`	irradiance marker
type	abbreviation for `<` *type* `..>`
mode	abbreviation for `<.` *mode* `.>`
handle	abbreviation for `<..` *handle* `>`
`I`	abbreviation for `<I.>`
`.`	match anything (in context)
`[` *A* ... `]`	match any element in set
`[^`*A* `]`	match all but *A*
AB	*A* followed by *B*
A`|`*B*	*A* or *B*
A`?`	zero or one *A*
A`*`	zero or more *A*s
A`+`	one or more *A*s
A`{`*n*`}`	a sequence of *n* *A*s
A`{`*n*`,`*m*`}`	*n* to *m* occurences of *A*
A`{`*n*`,}`	equivalent to *A*`{`*n*`}`*A*`*`
`(` ... `)`	grouping
`^` *expression*	complement of *expression*
expression-1 `&` *expression-2*	match both expressions
name`:` *expression*	assign *expression* to *name*
`$`*name*	use value of *name*

19 Bibliography

T. Driemeyer, *Rendering with mental ray*, 3nd edn. Springer, Wien New York, 2005 (mental ray handbooks, vol. 1).

Paul Heckbert. "Adaptive Radiosity Textures for Bidirectional Ray Tracing." *Computer Graphics* (SIGGRAPH '90 Proceedings), vol. 24, no. 4, August 1990, pp. 145–154.

ANSI, *ANSI Approved Standard File Format for Electronic Transfer of Photometric Data*, ANSI/IES LM-63-02.

NVIDIA Corporation, Santa Clara, California, United States. *NVIDIA Iray® API reference*.

NVIDIA Corporation, Santa Clara, California, United States. *NVIDIA Material Definition Language: Technical Introduction*. Version 1.1

NVIDIA Corporation, Santa Clara, California, United States. *NVIDIA Material Definition Language: Language Specification*. Version 1.1

mental images GmbH, Berlin, Germany. *mental ray® Manual, version 3.9*. Online Documentation.

Adobe Systems Incorporated, San Jose, USA. *Adobe Photoshop File Formats Specification*,[1] June 2012.

Berners-Lee, et. al., *Uniform Resource Identifier (URI): Generic Syntax*, RFC 2396, August 1988.

Adobe Systems Incorporated, San Jose, USA. *RTMP Command Messages*, April 2009.

Adobe Systems Incorporated, San Jose, USA. *RTMP Message Formats*, April 2009.

Adobe Systems Incorporated, San Jose, USA. *RTMP Chunk Stream*, April 2009.

Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD Thesis, Stanford University, 1997.

Johannes Jendersie. *Fast Spectral Upsampling of Volume Attenuation*. Ray Tracing Gems 2, 2021.

1. `https://www.adobe.com/devnet-apps/photoshop/fileformatashtml/`

20 Source code for examples

20.1 example_bridge_client.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>
#include <sstream>
#include <string>

#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"


std::string bytes_to_string( mi::Float64 size)
{
  char buffer[256];
  if( size > 1048576.0)
    snprintf( buffer, sizeof( buffer)-1, "%.1f MB", size/1048576.0);
  else if( size > 1024)
    snprintf( buffer, sizeof( buffer)-1, "%.1f kB", size/1024.0);
  else
    snprintf( buffer, sizeof( buffer)-1, "%d bytes", static_cast<mi::Sint32>( size));
  return buffer;
}

class Progress_callback
 : public mi::base::Interface_implement<mi::neuraylib::IProgress_callback>
{
public:
  void progress( mi::Float64 value, const char* area, const char* message)
  {
    if( strcmp( area, "bridge_bytes_uploaded") == 0)
      m_bridge_bytes_uploaded = value;
    else if( strcmp( area, "bridge_pending_cache_status") == 0)
      m_bridge_pending_cache_status = value;
    else if( strcmp( area, "bridge_pending_data_serialization") == 0)
      m_bridge_pending_data_serialization = value;
    else if( strcmp( area, "bridge_pending_hash_calculations") == 0)
      m_bridge_pending_hash_calculations = value;
    else if( strcmp( area, "bridge_total_bytes_to_upload") == 0)
      m_bridge_total_bytes_to_upload = value;
    else if( strcmp( area, "bridge_updated_elements") == 0)
      m_bridge_updated_elements = value;
    else if( strcmp( area, "bridge_upload_state") == 0)
```

```
    m_bridge_upload_state = value;
  else if( strcmp( area, "bridge_uploaded_element_bytes") == 0)
    m_uploaded_element_uploaded_bytes = value;
  else if( strcmp( area, "bridge_uploaded_element") == 0) {
    if( message) {
      std::stringstream s;
      s << "\"" << message << "\"";
      m_uploaded_element_name = s.str();
    } else
      m_uploaded_element_name = "unnamed element";
    m_uploaded_element_size = value;
  } else
    return; // just print Bridge-related progress messages

  if( m_bridge_upload_state == 0) {

  fprintf( stderr, "Bridge detecting changes: %llu modified element(s) detected.\n",
      static_cast<mi::Size>( m_bridge_updated_elements));

  } else if( m_bridge_upload_state == 1) {

    fprintf( stderr, "Bridge calculating hashes: %llu pending calculation(s), "
      "%s uploaded.\n",
      static_cast<mi::Size>( m_bridge_pending_hash_calculations),
      bytes_to_string( m_bridge_bytes_uploaded).c_str());

  } else if( m_bridge_upload_state == 2) {

    fprintf( stderr, "Bridge querying cache status: %llu pending request(s), "
      "%s uploaded.\n",
      static_cast<mi::Size>( m_bridge_pending_cache_status),
      bytes_to_string( m_bridge_bytes_uploaded).c_str());

  } else if( m_bridge_upload_state == 3) {

    if( m_bridge_pending_data_serialization > 0.0)
    fprintf( stderr, "Bridge upload: %s/%s (%.0lf%%) total - data serialization for "
        "%llu element(s) pending.\n",
        bytes_to_string( m_bridge_bytes_uploaded).c_str(),
        bytes_to_string( m_bridge_total_bytes_to_upload).c_str(),
        (m_bridge_bytes_uploaded / m_bridge_total_bytes_to_upload * 100.0),
        static_cast<mi::Size>( m_bridge_pending_data_serialization));
    else
   fprintf( stderr, "Bridge upload: %s/%s (%.0lf%%) total - %s/%s (%.0lf%%) for %s.\n",
        bytes_to_string( m_bridge_bytes_uploaded).c_str(),
        bytes_to_string( m_bridge_total_bytes_to_upload).c_str(),
        (m_bridge_bytes_uploaded / m_bridge_total_bytes_to_upload * 100.0),
        bytes_to_string( m_uploaded_element_uploaded_bytes).c_str(),
        bytes_to_string( m_uploaded_element_size).c_str(),
        (m_uploaded_element_uploaded_bytes / m_uploaded_element_size * 100.0),
        m_uploaded_element_name.c_str());

  } else if( m_bridge_upload_state == 4) {

   fprintf( stderr, "Bridge waiting for server to finish processing the upload.\n");

  } else if( m_bridge_upload_state == 5) {
```

```
      fprintf( stderr, "Bridge upload completed.\n");
      m_bridge_upload_state = -1;

    }
  }

private:
  mi::Float64 m_bridge_bytes_uploaded;
  mi::Float64 m_bridge_pending_cache_status;
  mi::Float64 m_bridge_pending_data_serialization;
  mi::Float64 m_bridge_pending_hash_calculations;
  mi::Float64 m_bridge_total_bytes_to_upload;
  mi::Float64 m_bridge_updated_elements;
  mi::Float64 m_bridge_upload_state;
  std::string m_uploaded_element_name;
  mi::Float64 m_uploaded_element_size;
  mi::Float64 m_uploaded_element_uploaded_bytes;
};

class Iray_bridge_snapshot_callback
 : public mi::base::Interface_implement<mi::bridge::IIray_bridge_snapshot_callback>
{
public:
  void ready( mi::Sint32 error_code, const char* /*file_name*/)
  {
    if( error_code == 0)
      fprintf( stderr, "Successfully created cb snapshot.\n");
    else
      fprintf( stderr, "Failed to create cb snapshot.\n");
    m_condition.signal();
  }

  void progress( mi::Float64 value, const char* area, const char* message)
  {
    fprintf( stderr, "Progress: %.4f %s %s\n", value, area, message);
  }

  void wait_for_ready_callback()
  {
    m_condition.wait();
  }

private:
  mi::base::Condition m_condition;
};

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Bridge client, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
```

```
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
 check_success( pc->load_plugin_library( "iray_bridge_client" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void import_and_store_scene( mi::neuraylib::INeuray* neuray, const char* scene_file)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope( database->get_global_scope());
 mi::base::Handle<mi::neuraylib::ITransaction> transaction( scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
 mi::base::Handle<const mi::IString> uri( import_api->convert_filename_to_uri( scene_file));
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), uri->get_c_str()));
  check_success( import_result->get_error_number() == 0);

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  check_success( scene.is_valid_interface());
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());

  // And store it in the database
  transaction->store( scene.get(), "the_scene");
  scene = 0;
  transaction->commit();
}

void rendering(
 mi::neuraylib::INeuray* neuray, const char* bridge_server_url, const char* security_token)
{
  // Configure the Iray Bridge client
  mi::base::Handle<mi::bridge::IIray_bridge_client> iray_bridge_client(
    neuray->get_api_component<mi::bridge::IIray_bridge_client>());
  check_success( iray_bridge_client.is_valid_interface());
  check_success( iray_bridge_client->set_application_url( bridge_server_url) == 0);
  check_success( iray_bridge_client->set_security_token( security_token) == 0);

 // Get the database, the global scope of the database, and create a transaction in the global
  // scope.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope( database->get_global_scope());
 mi::base::Handle<mi::neuraylib::ITransaction> transaction( scope->create_transaction());
```

```
  check_success( transaction.is_valid_interface());

  // Create the render context using the iray_cloud render mode.
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->edit<mi::neuraylib::IScene>( "the_scene"));
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray_cloud"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  mi::base::Handle<mi::IString> video_format( transaction->create<mi::IString>());
  video_format->set_c_str( "jpg");
  render_context->set_option( "video_format", video_format.get());

  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
  mi::base::Handle<mi::neuraylib::IProgress_callback> callback(
    new Progress_callback());
  render_context->render( transaction.get(), render_target.get(), callback.get());

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_bridge_client.png", canvas.get());

  transaction->commit();
}

void make_snapshot( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  mi::base::Handle<mi::neuraylib::IScope> scope( database->get_global_scope());
 mi::base::Handle<mi::neuraylib::ITransaction> transaction( scope->create_transaction());

  // Create a context for the snapshot.
  mi::base::Handle<mi::bridge::IIray_bridge_client> iray_bridge_client(
    neuray->get_api_component<mi::bridge::IIray_bridge_client>());
  mi::base::Handle<mi::bridge::IIray_bridge_snapshot_context> snapshot_context(
    iray_bridge_client->create_snapshot_context( transaction.get(), "the_scene"));

 // Create a callback instance, trigger snapshot creation, and wait for the callback to signal
  // completion of the snapshot.
 mi::base::Handle<Iray_bridge_snapshot_callback> callback( new Iray_bridge_snapshot_callback());
  mi::Sint32 result
   = snapshot_context->create_snapshot( transaction.get(), "snapshot.cb", callback.get());
  if( result >= 0)
```

```
    callback->wait_for_ready_callback();

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 5) {
   std::cerr << "Usage: example_bridge_client <scene_file> <mdl_path> <bridge_url> \\\n"
          "           <security_token>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* scene_file        = argv[1];
  const char* mdl_path          = argv[2];
  const char* bridge_server_url = argv[3];
  const char* security_token    = argv[4];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Import the scene into the DB
  import_and_store_scene( neuray.get(), scene_file);

 // Upload scene and render on the Bridge server, then export the rendered image to disk
  rendering( neuray.get(), bridge_server_url, security_token);

  // Make a snapshot of the scene on the Bridge server
  make_snapshot( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.2 example_bridge_server.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include <mi/neuraylib.h>

#include "example_shared.h"

class Application_session_handler
 : public mi::base::Interface_implement<mi::bridge::IApplication_session_handler>
{
public:
  Application_session_handler( const std::string& security_token)
   : m_security_token( security_token) { }

  bool on_session_connect( mi::bridge::IServer_session* session)
  {
    const char* security_token = session->get_security_token();
    return security_token && m_security_token == security_token;

  }
private:
  std::string m_security_token;
};

void run_bridge_server(
  mi::neuraylib::INeuray* neuray,
  const char* disk_cache_path,
  const char* snapshot_path,
  const char* bridge_server_address,
  const char* application_path,
  const char* security_token,
  const char* ssl_cert_file,
  const char* ssl_private_key_file,
  const char* ssl_password)
{
  // Start the HTTP server handling the Bridge connection using WebSockets.
  mi::base::Handle<mi::http::IFactory> http_factory(
    neuray->get_api_component<mi::http::IFactory>());
  mi::base::Handle<mi::http::IServer> http_server( http_factory->create_server());
  if( ssl_cert_file && ssl_private_key_file && ssl_password)
    http_server->start_ssl(
      bridge_server_address, ssl_cert_file, ssl_private_key_file, ssl_password);
  else
    http_server->start( bridge_server_address);

  // Access the API component for the Iray Bridge server and create and application.
  mi::base::Handle<mi::bridge::IIray_bridge_server> iray_bridge_server(
    neuray->get_api_component<mi::bridge::IIray_bridge_server>());
  check_success( iray_bridge_server.is_valid_interface());
  mi::base::Handle<mi::bridge::IIray_bridge_application> iray_bridge_application(
    iray_bridge_server->create_application( application_path, http_server.get()));
```

```
  if( iray_bridge_application.is_valid_interface()) {

    // Configure the application.
    check_success( iray_bridge_application->set_disk_cache( disk_cache_path) == 0);
    check_success( iray_bridge_application->set_snapshot_path( snapshot_path) == 0);
   mi::base::Handle<mi::bridge::IApplication_session_handler> application_session_handler(
      new Application_session_handler( security_token));
    check_success( iray_bridge_application->set_session_handler(
      application_session_handler.get()) == 0);

    // Run the Iray Bridge server for a fixed time span.
    iray_bridge_application->open();
    sleep_seconds( 60);
    iray_bridge_application->close();
  }

  http_server->shutdown();
}

void configuration( mi::neuraylib::INeuray* neuray)
{
  // Load the OpenImageIO, Iray Photoreal, and Iray Bridge server plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
 check_success( pc->load_plugin_library( "iray_bridge_server" MI_BASE_DLL_FILE_EXT) == 0);
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 6 && argc != 9) {
   std::cerr << "Usage: example_bridge_server <disk_cache_path> <snapshot_path> \\\n"
       << "         <bridge_server_address> <application_path> <security_token> \\\n"
         << "           [<ssl_cert_file> <ssl_private_key_file> <ssl_password>]"
         << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* disk_cache_path       = argv[1];
  const char* snapshot_path         = argv[2];
  const char* bridge_server_address = argv[3];
  const char* application_path      = argv[4];
  const char* security_token        = argv[5];
  const char* ssl_cert_file         = argc >= 9 ? argv[6] : 0;
  const char* ssl_private_key_file  = argc >= 9 ? argv[7] : 0;
  const char* ssl_password          = argc >= 9 ? argv[8] : 0;

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get());
```

```
  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Listen to Bridge clients
 run_bridge_server( neuray.get(), disk_cache_path, snapshot_path, bridge_server_address,
   application_path, security_token, ssl_cert_file, ssl_private_key_file, ssl_password);

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.3 example_configuration.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"

class Logger : public mi::base::Interface_implement<mi::base::ILogger>
{
public:
  void message(
    mi::base::Message_severity level, const char* module_category,
    const mi::base::Message_details&, const char* message)
  {
    const char* log_level = get_log_level( level);
    fprintf( stderr, "Log level = '%s', module:category = '%s', message = '%s'\n",
      log_level, module_category, message);

   // Do not return in case of fatal log messages since the process will be terminated anyway.
    // In this example just call exit().
    if( level == mi::base::MESSAGE_SEVERITY_FATAL)
      exit( EXIT_FAILURE);
  }

private:
  const char* get_log_level( mi::base::Message_severity level)
  {
    switch( level) {
      case mi::base::MESSAGE_SEVERITY_FATAL:
        return "FATAL";
      case mi::base::MESSAGE_SEVERITY_ERROR:
        return "ERROR";
      case mi::base::MESSAGE_SEVERITY_WARNING:
        return "WARNING";
      case mi::base::MESSAGE_SEVERITY_INFO:
        return "INFO";
      case mi::base::MESSAGE_SEVERITY_VERBOSE:
        return "VERBOSE";
      case mi::base::MESSAGE_SEVERITY_DEBUG:
        return "DEBUG";
      default:
        return "";
    }
  }
};

int main( int /*argc*/, char* /*argv*/[])
{
  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Create an instance of our logger
```

```
  mi::base::Handle<mi::base::ILogger> logger( new Logger());

  // Open new block to ensure that all handles except neuray went out of scope
  // when calling shutdown() after the end of this block.
  {

    // Configure the neuray library before startup. Here we set our own logger.
    check_success( neuray->get_status() == mi::neuraylib::INeuray::PRE_STARTING);
    mi::base::Handle<mi::neuraylib::ILogging_configuration> logging_configuration(
      neuray->get_api_component<mi::neuraylib::ILogging_configuration>());
    logging_configuration->set_receiving_logger( logger.get());

    // other configuration settings go here, none in this example

    // Start the neuray library
    mi::Sint32 result = neuray->start();
    check_start_success( result);
    check_success( neuray->get_status() == mi::neuraylib::INeuray::STARTED);

    // scene graph manipulations and rendering calls go here, none in this example.
  }

  // Shut down the library
  check_success( neuray->shutdown() == 0);
  check_success( neuray->get_status() == mi::neuraylib::INeuray::SHUTDOWN);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.4 example_exporter.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include <mi/neuraylib.h>

#include "example_shared.h"

#include "vanilla_exporter.h"

mi::base::Handle<mi::neuraylib::IExporter> exporter;

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rendering_configuration(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rendering_configuration->add_mdl_path( mdl_path) == 0);
  check_success( rendering_configuration->add_mdl_path( ".") == 0);

  // Register the Vanilla exporter.
  mi::base::Handle<mi::neuraylib::IExtension_api> extension_api(
    neuray->get_api_component<mi::neuraylib::IExtension_api>());
  check_success( extension_api.is_valid_interface());
  exporter = new Vanilla_exporter;
  check_success( extension_api->register_exporter( exporter.get()) == 0);

  // Load the .mi importer plugin.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> plugin_configuration(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
  check_success( plugin_configuration->load_plugin_library(
    "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void test_exporter( mi::neuraylib::INeuray* neuray, const char* scene_file)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
 mi::base::Handle<const mi::IString> uri( import_api->convert_filename_to_uri( scene_file));
```

```
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), uri->get_c_str()));
  check_success( import_result->get_error_number() == 0);
  const char* root_group = import_result->get_rootgroup();

  // Export the scene to a file test3.vnl (implicitly using the Vanilla exporter).
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IExport_result> export_result(
    export_api->export_scene( transaction.get(), "file:test3.vnl", root_group));
  check_success( export_result.is_valid_interface());

  // Print all messages
  for( mi::Size i = 0; i < export_result->get_messages_length(); ++i)
    std::cout << export_result->get_message( i) << std::endl;

  check_success( export_result->get_error_number() == 0);
  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 3) {
    std::cerr << "Usage: example_exporter <scene_file> <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* scene_file = argv[1];
  const char* mdl_path   = argv[2];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Test the Vanilla exporter
  test_exporter( neuray.get(), scene_file);

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);

  // Unregister the Vanilla exporter.
  mi::base::Handle<mi::neuraylib::IExtension_api> extension_api(
    neuray->get_api_component<mi::neuraylib::IExtension_api>());
  check_success( extension_api->unregister_exporter( exporter.get()) == 0);
  exporter = 0;
  extension_api = 0;
  neuray = 0;
```

```
  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.5 example_fibers.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>
#include <map>
#include <vector>

unsigned int lcg_a    = 134775815;
unsigned int lcg_c    = 123456;
unsigned int lcg_m    = 1 << 31;
unsigned int lcg_prev = 33478921;

unsigned int get_random_uint()
{
  mi::Uint64 val = (lcg_a * lcg_prev + lcg_c) % lcg_m;
  lcg_prev = static_cast<unsigned int>(val);
  return lcg_prev;
}

float get_random_float(const float& min, const float& max)
{
  return ((float(get_random_uint()) * (max - min)) / float(lcg_m)) + min;
}

mi::neuraylib::IFibers* create_fibers_ball( mi::neuraylib::ITransaction* transaction)
{
  // Some constants to define the fibers ball
  const mi::Float32_3 sphere_center(0.f, 0.f, 0.f);
  const mi::Float32   sphere_radius = 0.2f;
  const mi::Float32   min_length    = 0.8f;
  const mi::Float32   max_length    = 1.0f;
  const mi::Float32   base_radius   = 0.01f;
  const mi::Float32   tip_radius    = 0.002f;
  const mi::Float32   randomness    = 0.2f;
  const unsigned int  num_fibers    = 1000;
  const unsigned int  num_vertices_per_fiber = 9; // must be at least 4

 mi::neuraylib::IFibers *fibers = transaction->create<mi::neuraylib::IFibers>("Fibers");
  fibers->set_type(mi::neuraylib::FIBER_TYPE_BSPLINE);

 // the fibers will have an implicit U mapping from 0.0 to 1.0 along them from base to tip

  std::vector<mi::Float32_4> control_points;
  control_points.reserve(num_vertices_per_fiber);

  for (unsigned int n=0; n<num_fibers; ++n) {
   // draw a random position on a sphere of radius 'sphere_radius' and center 'sphere_center'
    // generating random direction vectors
```

```
  const float rx = get_random_float(-1.0f, 1.0f);
  const float ry = get_random_float(-1.0f, 1.0f);
  const float rz = get_random_float(-1.0f, 1.0f);
  mi::Float32_3 dir(rx, ry, rz);
  dir.normalize();

  // first vertex of the fiber
  const mi::Float32_3 first_pos = sphere_center + sphere_radius * dir;

  // compute tangent space around the direction
mi::Float32_3 tu = (dir.y * dir.y > dir.x * dir.x) ? mi::Float32_3(0.0f, dir.z, -dir.y) : mi::Float3
  mi::Float32_3 tv = cross(tu, dir);
  tv.normalize();
  tu = cross(tv, dir);
  tu.normalize();

// compute a random (approximate) length of the fiber, between 'min_length' and 'max_length'
  const mi::Float32 fiber_length = get_random_float(min_length, max_length);

  // (approximate) length of a segment of the fiber
 const mi::Float32 segment_step = fiber_length / float(num_vertices_per_fiber - 1);

  // step for the radius
  const mi::Float32 radius_step = (tip_radius - base_radius) /
    float(num_vertices_per_fiber - 1);

  control_points.clear();

  // first compute all the normal B-spline control points along with their radii
  // and save them in 'control_points' and 'radii'
  for (unsigned int k = 0; k < num_vertices_per_fiber; ++k) {
    mi::Float32_3 v = first_pos + float(k) * segment_step * dir;

    if (k > 0) {
      // all vertices get randomly moved in the tangent space of the direction
      // except the first one, which is the base vertex of the fiber

      const float du = get_random_float(-randomness*0.5f, randomness*0.5f);
      const float dv = get_random_float(-randomness*0.5f, randomness*0.5f);
      v += (du * tu + dv * tv);
    }

    const mi::Float32 radius = base_radius + float(k) * radius_step;

    control_points.push_back(mi::Float32_4(v.x, v.y, v.z, radius));
  }

  // compute an extra point to be put at the very beginning of the B-spline,
  // this is a ghost point (or phantom point) that has the effect of making
  // the curve effectively start at 'control_points[0]'
  mi::Float32_4 ghost_point = (2.0 * control_points[0]) - control_points[1];
  mi::Float32   ghost_radius = control_points[0].w;

// add a fiber that has the ghost point at the beginning and all the computed control points
const mi::neuraylib::Fiber_handle_struct fh = fibers->add_fiber(num_vertices_per_fiber + 1);

fibers->set_control_point(fh, 0, mi::Float32_3(ghost_point.x, ghost_point.y, ghost_point.z));
```

```
    fibers->set_radius(fh, 0, ghost_radius);

    // add control points/radii
    fibers->set_fiber_data(fh, 1, control_points.data(), control_points.size());
  }
  return fibers;
}

void setup_scene( mi::neuraylib::ITransaction* transaction, const char* rootgroup)
{
  // Create the fibers object
 mi::base::Handle<mi::neuraylib::IFibers> fibers_object(create_fibers_ball( transaction));
  transaction->store( fibers_object.get(), "fibers_obj");

  // Create the instance for the fibers object
  mi::base::Handle<mi::neuraylib::IInstance> instance(
    transaction->create<mi::neuraylib::IInstance>( "Instance"));
  instance->attach( "fibers_obj");

  // Set the transformation matrix, the visible attribute, and the material
  mi::Float64_4_4 matrix( 1.0 );
  matrix.translate( 0.0, -0.4, 0.0);
  instance->set_matrix( matrix );

  mi::base::Handle<mi::IBoolean> visible(
    instance->create_attribute<mi::IBoolean>( "visible", "Boolean"));
  visible->set_value( true);

 mi::base::Handle<mi::IRef> material( instance->create_attribute<mi::IRef>( "material", "Ref"));
  material->set_reference( "unicorn_hair");

  transaction->store( instance.get(), "instance_fibers");

  // Attach the instance to the root group
  mi::base::Handle<mi::neuraylib::IGroup> group(
    transaction->edit<mi::neuraylib::IGroup>( rootgroup));
  group->attach( "instance_fibers");
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray)
{
```

```
  // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Add two triangle meshes to the scene
  setup_scene( transaction.get(), import_result->get_rootgroup());

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_fibers.png", canvas.get());

  transaction->commit();
}
```

```
int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_fibers <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.6 example_freeform_surface.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>

mi::neuraylib::IFreeform_surface* create_sphere(
  mi::neuraylib::ITransaction* transaction, mi::neuraylib::Basis_type basis)
{
  const mi::Size n_points_u = 9;
  const mi::Size n_points_v = 5;

  // The control points
  mi::Float32_3 control_points[n_points_u * n_points_v] = {
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),
    mi::Float32_3(  0.0f,  0.0f, -1.0f),

    mi::Float32_3(  1.0f,  0.0f, -1.0f),
    mi::Float32_3(  1.0f,  1.0f, -1.0f),
    mi::Float32_3(  0.0f,  1.0f, -1.0f),
    mi::Float32_3( -1.0f,  1.0f, -1.0f),
    mi::Float32_3( -1.0f,  0.0f, -1.0f),
    mi::Float32_3( -1.0f, -1.0f, -1.0f),
    mi::Float32_3(  0.0f, -1.0f, -1.0f),
    mi::Float32_3(  1.0f, -1.0f, -1.0f),
    mi::Float32_3(  1.0f,  0.0f, -1.0f),

    mi::Float32_3(  1.0f,  0.0f,  0.0f),
    mi::Float32_3(  1.0f,  1.0f,  0.0f),
    mi::Float32_3(  0.0f,  1.0f,  0.0f),
    mi::Float32_3( -1.0f,  1.0f,  0.0f),
    mi::Float32_3( -1.0f,  0.0f,  0.0f),
    mi::Float32_3( -1.0f, -1.0f,  0.0f),
    mi::Float32_3(  0.0f, -1.0f,  0.0f),
    mi::Float32_3(  1.0f, -1.0f,  0.0f),
    mi::Float32_3(  1.0f,  0.0f,  0.0f),

    mi::Float32_3(  1.0f,  0.0f,  1.0f),
    mi::Float32_3(  1.0f,  1.0f,  1.0f),
    mi::Float32_3(  0.0f,  1.0f,  1.0f),
    mi::Float32_3( -1.0f,  1.0f,  1.0f),
    mi::Float32_3( -1.0f,  0.0f,  1.0f),
```

```
    mi::Float32_3( -1.0f, -1.0f,  1.0f),
    mi::Float32_3(  0.0f, -1.0f,  1.0f),
    mi::Float32_3(  1.0f, -1.0f,  1.0f),
    mi::Float32_3(  1.0f,  0.0f,  1.0f),

    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f),
    mi::Float32_3(  0.0f,  0.0f,  1.0f)
  };

  // Create an empty freeform surface with one surface
  mi::neuraylib::IFreeform_surface* sphere
    = transaction->create<mi::neuraylib::IFreeform_surface>( "Freeform_surface");
  mi::base::Handle<mi::neuraylib::ISurface> surface( sphere->add_surface());

 // There are two choices for the basis used to parameterize the freeform surface: bezier and
 // b-spline. The other data depends on that choice, except for the control points which are
  // identical (for this specific case).

  if( basis == mi::neuraylib::BASIS_BEZIER) {

    // Set up the basis type, degrees, and number of patches
    surface->set_basis_type( mi::neuraylib::BASIS_BEZIER);
    surface->set_degree( mi::neuraylib::DIMENSION_U, 2);
    surface->set_degree( mi::neuraylib::DIMENSION_V, 2);
    surface->set_patches_size( mi::neuraylib::DIMENSION_U, 4);
    surface->set_patches_size( mi::neuraylib::DIMENSION_V, 2);

    // Set up the parameter vectors
    surface->set_parameter( mi::neuraylib::DIMENSION_U, 0, 0.0f);
    surface->set_parameter( mi::neuraylib::DIMENSION_U, 1, 0.25f);
    surface->set_parameter( mi::neuraylib::DIMENSION_U, 2, 0.5f);
    surface->set_parameter( mi::neuraylib::DIMENSION_U, 3, 0.75f);
    surface->set_parameter( mi::neuraylib::DIMENSION_U, 4, 1.0f);
    surface->set_parameter( mi::neuraylib::DIMENSION_V, 0, 0.0f);
    surface->set_parameter( mi::neuraylib::DIMENSION_V, 1, 0.5f);
    surface->set_parameter( mi::neuraylib::DIMENSION_V, 2, 1.0f);

    // The weights
    mi::Float32 weights[n_points_u * n_points_v] = {
      1.0f, 1.0f, 2.0f, 1.0f, 1.0f, 1.0f, 2.0f, 1.0f, 1.0f,
      1.0f, 1.0f, 2.0f, 1.0f, 1.0f, 1.0f, 2.0f, 1.0f, 1.0f,
      2.0f, 2.0f, 4.0f, 2.0f, 2.0f, 2.0f, 4.0f, 2.0f, 2.0f,
      1.0f, 1.0f, 2.0f, 1.0f, 1.0f, 1.0f, 2.0f, 1.0f, 1.0f,
      1.0f, 1.0f, 2.0f, 1.0f, 1.0f, 1.0f, 2.0f, 1.0f, 1.0f
    };

    // Set up the control points and weights
    for( mi::Uint32 i = 0; i < n_points_u; ++i)
      for( mi::Uint32 j = 0; j < n_points_v; ++j) {
        surface->set_control_point( i, j, control_points[j * n_points_u + i]);
```

```
      surface->set_weight( i, j, weights[j * n_points_u + i]);
    }

} else {

  // Set up the basis type, degrees, and number of patches
  surface->set_basis_type( mi::neuraylib::BASIS_BSPLINE);
  surface->set_degree( mi::neuraylib::DIMENSION_U, 2);
  surface->set_degree( mi::neuraylib::DIMENSION_V, 2);
  surface->set_patches_size( mi::neuraylib::DIMENSION_U, 7);
  surface->set_patches_size( mi::neuraylib::DIMENSION_V, 3);

  // Set up the parameter vectors
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  0, 0.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  1, 0.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  2, 0.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  3, 0.25f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  4, 0.25f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  5, 0.5f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  6, 0.5f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  7, 0.75f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  8, 0.75f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U,  9, 1.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U, 10, 1.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_U, 11, 1.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  0, 0.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  1, 0.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  2, 0.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  3, 0.5f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  4, 0.5f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  5, 1.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  6, 1.0f);
  surface->set_parameter( mi::neuraylib::DIMENSION_V,  7, 1.0f);

  // The weights
  mi::Float32 w = 0.5f * std::sqrt( 2.0f);
  mi::Float32 weights[n_points_u * n_points_v] = {
    1.0f,    w, 1.0f,    w, 1.0f,    w, 1.0f,    w, 1.0f,
      w, 0.5f,    w, 0.5f,    w, 0.5f,    w, 0.5f,    w,
    1.0f,    w, 1.0f,    w, 1.0f,    w, 1.0f,    w, 1.0f,
      w, 0.5f,    w, 0.5f,    w, 0.5f,    w, 0.5f,    w,
    1.0f,    w, 1.0f,    w, 1.0f,    w, 1.0f,    w, 1.0f
  };

  // Set up the control points and weights
  for( mi::Uint32 i = 0; i < n_points_u; ++i)
    for( mi::Uint32 j = 0; j < n_points_v; ++j) {
      surface->set_control_point( i, j, control_points[j * n_points_u + i]);
      surface->set_weight( i, j, weights[j * n_points_u + i]);
    }
}

surface = 0;

// Set up the approximation granularity
mi::base::Handle<mi::IStructure> approx(
  sphere->create_attribute<mi::IStructure>( "approx", "Approx"));
```

```
  mi::base::Handle<mi::ISint8> method( approx->get_value<mi::ISint8>( "method"));
  method->set_value( 0);
  method = 0;
 mi::base::Handle<mi::IFloat32> const_u( approx->get_value<mi::IFloat32>( "const_u"));
  const_u->set_value( 12.0f);
  const_u = 0;
 mi::base::Handle<mi::IFloat32> const_v( approx->get_value<mi::IFloat32>( "const_v"));
  const_v->set_value( 12.0f);
  const_v = 0;
  approx = 0;

  return sphere;
}

void setup_scene( mi::neuraylib::ITransaction* transaction, const char* rootgroup)
{
  // Create the red sphere
  mi::base::Handle<mi::neuraylib::IFreeform_surface> mesh(
    create_sphere( transaction, mi::neuraylib::BASIS_BEZIER));
  transaction->store( mesh.get(), "mesh_red");

  // Create the instance for the red sphere
  mi::base::Handle<mi::neuraylib::IInstance> instance(
    transaction->create<mi::neuraylib::IInstance>( "Instance"));
  instance->attach( "mesh_red");

  // Set the transformation matrix, the visible attribute, and the material
  mi::Float64_4_4 matrix( 1.0);
  matrix.translate( -0.6, -0.5, 0.7);
 mi::Float64_4_4 matrix_scale( 2.0, 0, 0, 0, 0, 2.0, 0, 0, 0, 0, 2.0, 0, 0, 0, 0, 1.0);
  matrix *= matrix_scale;
  instance->set_matrix( matrix);

  mi::base::Handle<mi::IBoolean> visible(
    instance->create_attribute<mi::IBoolean>( "visible", "Boolean"));
  visible->set_value( true);

 mi::base::Handle<mi::IRef> material( instance->create_attribute<mi::IRef>( "material", "Ref"));
  material->set_reference( "red_material");

  transaction->store( instance.get(), "instance_red");

  // And attach the instance to the root group
  mi::base::Handle<mi::neuraylib::IGroup> group(
    transaction->edit<mi::neuraylib::IGroup>( rootgroup));
  group->attach( "instance_red");
  group = 0;

  // Create the blue sphere
  mesh = create_sphere( transaction, mi::neuraylib::BASIS_BSPLINE);

  // Trim the sphere to the part above the ground plane
  mi::base::Handle<mi::neuraylib::ISurface> surface(
    mesh->edit_surface( mi::neuraylib::Surface_handle( 0)));
  surface->set_range( mi::neuraylib::DIMENSION_U, 0.0f, 0.5f);
  surface = 0;
```

```
  transaction->store( mesh.get(), "mesh_blue");

  // Create the instance for the blue sphere
  instance = transaction->create<mi::neuraylib::IInstance>( "Instance");
  instance->attach( "mesh_blue");

  // Set the transformation matrix, the visible attribute, and the material
  matrix = mi::Float64_4_4( 1.0);
  matrix.translate( -0.6, 0.0, -1.1);
 matrix_scale = mi::Float64_4_4( 2.0, 0, 0, 0, 0, 1.0, 0, 0, 0, 0, 2.0, 0, 0, 0, 0, 1.0);
  matrix *= matrix_scale;
  instance->set_matrix( matrix);

  visible = instance->create_attribute<mi::IBoolean>( "visible", "Boolean");
  visible->set_value( true);

  material = instance->create_attribute<mi::IRef>( "material", "Ref");
  material->set_reference( "blue_material");

  transaction->store( instance.get(), "instance_blue");

  // And attach the instance to the root group
  group = transaction->edit<mi::neuraylib::IGroup>( rootgroup);
  group->attach( "instance_blue");
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
```

```
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Add two instances of freeform surfaces
  setup_scene( transaction.get(), import_result->get_rootgroup());

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");
  scene = 0;

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_freeform_surface.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_freeform_surface <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
```

```
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.7 example_http_server.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

class Request_handler : public mi::base::Interface_implement<mi::http::IRequest_handler>
{
public:
  // Constructor. Stores the passed buffer.
  Request_handler( mi::neuraylib::IBuffer* buffer)
   : m_buffer( buffer, mi::base::DUP_INTERFACE) { }

  // Send buffer contents over the connection.
  bool handle( mi::http::IConnection* connection)
  {
    connection->enqueue( m_buffer.get());
    return true;
  }
private:
  // The stored buffer
  mi::base::Handle<mi::neuraylib::IBuffer> m_buffer;
};

class Response_handler : public mi::base::Interface_implement<mi::http::IResponse_handler>
{
public:
  void handle( mi::http::IConnection* connection)
  {
    mi::http::IResponse* iresponse( connection->get_response());
    iresponse->set_header( "Content-Type", "image/jpeg");
  }
};

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}
```

```
mi::base::Handle<mi::neuraylib::IBuffer> rendering(
  mi::neuraylib::INeuray* neuray, const char* scene_file)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
 mi::base::Handle<const mi::IString> uri( import_api->convert_filename_to_uri( scene_file));
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), uri->get_c_str()));
  check_success( import_result->get_error_number() == 0);

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Access the first canvas of the render target
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));

  // Convert content of the canvas to a JPG image
  mi::base::Handle<mi::neuraylib::IBuffer> buffer(
    image_api->create_buffer_from_canvas( canvas.get(), "jpg", "Rgb", "100"));

  transaction->commit();
```

```
  return buffer;
}

void run_http_server(
  mi::neuraylib::INeuray* neuray, mi::neuraylib::IBuffer* buffer, const char* port)
{
  // Create a server instance
  mi::base::Handle<mi::http::IFactory> http_factory(
    neuray->get_api_component<mi::http::IFactory>());
  mi::base::Handle<mi::http::IServer> http_server(
    http_factory->create_server());

  // Install our request and response handlers
 mi::base::Handle<mi::http::IRequest_handler> request_handler( new Request_handler( buffer));
  http_server->install( request_handler.get());
 mi::base::Handle<mi::http::IResponse_handler> response_handler( new Response_handler());
  http_server->install( response_handler.get());

  // Assemble server address
  std::string address = "0.0.0.0:";
  address += port;

  // Run server for fixed time interval
  http_server->start( address.c_str());
  sleep_seconds( 30);
  http_server->shutdown();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 4) {
   std::cerr << "Usage: example_http_server <scene_file> <mdl_path> <port>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* scene_file  = argv[1];
  const char* mdl_path    = argv[2];
  const char* port        = argv[3];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
 mi::base::Handle<mi::neuraylib::IBuffer> buffer = rendering( neuray.get(), scene_file);

  // Serve image via HTTP server on given port
  run_http_server( neuray.get(), buffer.get(), port);
```

```
  buffer = 0;

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.8 example_importer.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include "example_shared.h"

#include "vanilla_importer.h"

mi::base::Handle<mi::neuraylib::IImporter> importer;

void configuration( mi::neuraylib::INeuray* neuray)
{
  // Register the Vanilla importer.
  mi::base::Handle<mi::neuraylib::IExtension_api> extension_api(
    neuray->get_api_component<mi::neuraylib::IExtension_api>());
  check_success( extension_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IPlugin_api> plugin_api(
    neuray->get_api_component<mi::neuraylib::IPlugin_api>());
  check_success( plugin_api.is_valid_interface());
  importer = new Vanilla_importer( plugin_api.get());
  check_success( extension_api->register_importer( importer.get()) == 0);
}

void test_importer( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Prepare the importer options:
  // We do not want to have an additional prefix, but a list of all imported elements.
 mi::base::Handle<mi::IBoolean> list_elements( transaction->create<mi::IBoolean>( "Boolean"));
  list_elements->set_value( true);
 mi::base::Handle<mi::IMap> importer_options( transaction->create<mi::IMap>( "Map<Interface>"));
  importer_options->insert( "list_elements", list_elements.get());

  // Import the file test1.vnl (implicitly using the Vanilla importer).
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
   import_api->import_elements( transaction.get(), "file:test1.vnl", importer_options.get()));
  check_success( import_result.is_valid_interface());

  // Print all messages
```

```
  for( mi::Size i = 0; i < import_result->get_messages_length(); ++i)
    std::cout << import_result->get_message( i) << std::endl;

  // Print all imported elements
  for( mi::Size i = 0; i < import_result->get_elements_length(); ++i)
    std::cout << import_result->get_element( i) << std::endl;

  transaction->commit();
}

int main( int /*argc*/, char* /*argv*/[])
{
  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get());

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Test the Vanilla importer
  test_importer( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);

  // Unregister the Vanilla importer.
  mi::base::Handle<mi::neuraylib::IExtension_api> extension_api(
    neuray->get_api_component<mi::neuraylib::IExtension_api>());
  check_success( extension_api->unregister_importer( importer.get()) == 0);
  importer = 0;
  extension_api = 0;
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.9 example_mdl.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);
  check_success( rc->add_resource_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, .mi importer, and MDL distiller plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
 check_success( pc->load_plugin_library( "mdl_distiller" MI_BASE_DLL_FILE_EXT) == 0);
}

void render_and_export(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::ITransaction* transaction,
  const char* uri)
{
  // Create the render context using the Iray Photoreal render mode.
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->edit<mi::neuraylib::IScene>( "the_scene"));
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction, "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene.
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
  check_success( render_context->render( transaction, render_target.get(), 0) >= 0);

  // Write the image to disk.
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
```

```
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( uri, canvas.get());
}

void import_and_render_original_scene(
  mi::neuraylib::INeuray* neuray, mi::neuraylib::IScope* scope)
{
  // Create a transaction for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file.
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Create the scene object.
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  render_and_export( neuray, transaction.get(), "file:example_mdl_original.png");
  transaction->commit();
}

void modify_material_instance(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::IScope* scope,
  mi::neuraylib::IMdl_factory* mdl_factory)
{
  // Create a transaction and MDL expression factory.
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction.get()));

  {
    // Create a new argument for the "tint" parameter and set it to bright grey.
    mi::base::Handle<mi::neuraylib::IFunction_call> yellow_material(
      transaction->edit<mi::neuraylib::IFunction_call>( "yellow_material"));
    mi::base::Handle<const mi::neuraylib::IExpression_list> arguments(
      yellow_material->get_arguments());
    mi::base::Handle<const mi::neuraylib::IExpression> argument(
      arguments->get_expression( "tint"));
    mi::base::Handle<mi::neuraylib::IExpression> new_argument(
      expression_factory->clone( argument.get()));
    mi::base::Handle<mi::neuraylib::IExpression_constant> new_argument_constant(
```

```
      new_argument->get_interface<mi::neuraylib::IExpression_constant>());
   mi::base::Handle<mi::neuraylib::IValue> new_value( new_argument_constant->get_value());
    mi::math::Color bright_grey( 0.7f, 0.7f, 0.7f);
    mi::neuraylib::set_value( new_value.get(), bright_grey);

    // Set the new argument for the "tint" parameter of "yellow_material".
    check_success( yellow_material->set_argument( "tint", new_argument.get()) == 0);
  }
 // The same effect as in the preceding code block can alternatively be achieved by using the
  // helper class Argument_editor as follows:
  {
    // Set the "tint" argument of "yellow_material" to bright grey.
    mi::neuraylib::Argument_editor yellow_material(
      transaction.get(), "yellow_material", mdl_factory);
    mi::math::Color bright_grey( 0.7f, 0.7f, 0.7f);
    check_success( yellow_material.set_value( "tint", bright_grey) == 0);

  }

  render_and_export(
    neuray, transaction.get(), "file:example_mdl_modified_material_argument.png");
  transaction->commit();
}

void create_new_material_instance(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::IScope* scope,
  mi::neuraylib::IMdl_factory* mdl_factory)
{
  // Create a transaction, MDL value and expression factory.
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IValue_factory> value_factory(
    mdl_factory->create_value_factory( transaction.get()));
  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction.get()));

  {
   // Prepare the arguments for the new material instance: set the "tint" argument to white.
    mi::base::Handle<mi::neuraylib::IValue> value(
      value_factory->create_color( 1.0f, 1.0f, 1.0f));
    mi::base::Handle<mi::neuraylib::IExpression> expression(
      expression_factory->create_constant( value.get()));
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "tint", expression.get());

   // Create a material instance of the definition "mdl::main::diffuse_material" with the just
    // prepared arguments.
    mi::base::Handle<const mi::neuraylib::IFunction_definition> material_definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::main::diffuse_material(color)"));
    mi::Sint32 result;
    mi::base::Handle<mi::neuraylib::IFunction_call> material_instance(
      material_definition->create_function_call( arguments.get(), &result));
    check_success( result == 0);
```

```
    transaction->store( material_instance.get(), "white_material");

   // Attach the newly created material instance to the scene element "cube_instance", thereby
    // replacing the existing material instance "yellow_material".
    mi::base::Handle<mi::neuraylib::IInstance> instance(
      transaction->edit<mi::neuraylib::IInstance>( "cube_instance"));
   mi::base::Handle<mi::IArray> material( instance->edit_attribute<mi::IArray>( "material"));
    check_success(
    mi::set_value( material.get(), static_cast<mi::Size>( 0), "white_material") == 0);
  }

 render_and_export( neuray, transaction.get(), "file:example_mdl_new_material_instance.png");
  transaction->commit();
}

void attach_function_call(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::IScope* scope,
  mi::neuraylib::IMdl_factory* mdl_factory)
{
  // Create a transaction, MDL value and expression factory.
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IValue_factory> value_factory(
    mdl_factory->create_value_factory( transaction.get()));
  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction.get()));
  {
    // Create a DB element for the image and the texture referencing it.
    mi::base::Handle<mi::neuraylib::IImage> image(
      transaction->create<mi::neuraylib::IImage>( "Image"));
    check_success( image->reset_file( "nvidia_logo.png") == 0);
    transaction->store( image.get(), "nvidia_image");
    mi::base::Handle<mi::neuraylib::ITexture> texture(
      transaction->create<mi::neuraylib::ITexture>( "Texture"));
    texture->set_image( "nvidia_image");
   // Setting the gamma override value here is not strictly necessary, since the canvas of
    // the image has already a gamma value of 2.2. However, if new MDL source code is
   // generated based on this texture, the explicit setting here causes "::tex::gamma_srgb" to
    // be used instead of "::tex::gamma_default", which might be desired.
    texture->set_gamma( 2.2f);
    transaction->store( texture.get(), "nvidia_texture");
  }
  {
    // Import the "base.mdl" module.
    mi::base::Handle<mi::neuraylib::IImport_api> import_api(
      neuray->get_api_component<mi::neuraylib::IImport_api>());
    mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
      import_api->import_elements( transaction.get(), "${shader}/base.mdl"));
    check_success( import_result->get_error_number() == 0);
  }
  {
   // Lookup the exact name of the DB element for the MDL function "base::file_texture".
    mi::base::Handle<const mi::neuraylib::IModule> module(
      transaction->access<mi::neuraylib::IModule>( "mdl::base"));
    mi::base::Handle<const mi::IArray> overloads(
```

```
    module->get_function_overloads( "mdl::base::file_texture"));
  check_success( overloads->get_length() == 1);
  mi::base::Handle<const mi::IString> file_texture_name(
    overloads->get_element<mi::IString>( static_cast<mi::Uint32>( 0)));

 // Prepare the arguments of the function call for "mdl::base::file_texture": set the
  // "texture" argument to the "nvidia_texture" texture.
  mi::base::Handle<const mi::neuraylib::IFunction_definition> function_definition(
    transaction->access<mi::neuraylib::IFunction_definition>(
      file_texture_name->get_c_str()));
  mi::base::Handle<const mi::neuraylib::IType_list> types(
    function_definition->get_parameter_types());
  mi::base::Handle<const mi::neuraylib::IType> type(
    types->get_type( "texture"));
  mi::base::Handle<mi::neuraylib::IValue> value(
    value_factory->create( type.get()));
  check_success( mi::neuraylib::set_value( value.get(), "nvidia_texture") == 0);
  mi::base::Handle<mi::neuraylib::IExpression> expression(
    expression_factory->create_constant( value.get()));
  mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
    expression_factory->create_expression_list());
  arguments->add_expression( "texture", expression.get());

 // Create a function call from the definition "mdl::base::file_texture" with the just
  // prepared arguments.
  mi::Sint32 result;
  mi::base::Handle<mi::neuraylib::IFunction_call> function_call(
    function_definition->create_function_call( arguments.get(), &result));
  check_success( result == 0);
  function_definition = 0;
  transaction->store( function_call.get(), "file_texture_call");
}
{
 // Prepare the arguments of the function call for "mdl::base::texture_return.tint": set the
  // "s" argument to the "file_texture_call" function call.
  mi::base::Handle<mi::neuraylib::IExpression> expression(
    expression_factory->create_call( "file_texture_call"));
  mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
    expression_factory->create_expression_list());
  arguments->add_expression( "s", expression.get());

 // Create a function call from the definition "mdl::base::file_texture.tint" with the just
  // prepared arguments.
  mi::base::Handle<const mi::neuraylib::IFunction_definition> function_definition(
    transaction->access<mi::neuraylib::IFunction_definition>(
      "mdl::base::texture_return.tint(::base::texture_return)"));
  mi::Sint32 result;
  mi::base::Handle<mi::neuraylib::IFunction_call> function_call(
    function_definition->create_function_call( arguments.get(), &result));
  check_success( result == 0);
  transaction->store( function_call.get(), "texture_return.tint_call");
}
{
  // Create a new expression for the "tint" argument and set it to the
  // "texture_return.tint_call" function call.
  mi::base::Handle<mi::neuraylib::IExpression> expression(
    expression_factory->create_call( "texture_return.tint_call"));
```

```
    // Set the new argument for the "tint" parameter of "grey_material".
    mi::base::Handle<mi::neuraylib::IFunction_call> grey_material(
      transaction->edit<mi::neuraylib::IFunction_call>( "grey_material"));
    check_success( grey_material->set_argument( "tint", expression.get()) == 0);
  }
  // The same effect as in the preceding code block can alternatively be achieved by using the
  // helper class Mdl_argument_editor as follows:
  {
    // Attach "texture_return.tint_call" to the "tint" argument of "grey_material".
    mi::neuraylib::Argument_editor grey_material(
      transaction.get(), "grey_material", mdl_factory);
   check_success( grey_material.set_call( "tint", "texture_return.tint_call") == 0);
  }

 render_and_export( neuray, transaction.get(), "file:example_mdl_attached_function_call.png");
  transaction->commit();
}

void use_struct_and_array_constructors(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::IScope* scope,
  mi::neuraylib::IMdl_factory* mdl_factory)
{
  // Create a transaction, MDL value and expression factory.
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IType_factory> type_factory(
    mdl_factory->create_type_factory( transaction.get()));
  mi::base::Handle<mi::neuraylib::IValue_factory> value_factory(
    mdl_factory->create_value_factory( transaction.get()));
  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction.get()));

  {
    // Create a first color "color_layer0" layer using "texture_return.tint_call" and
    // "color_layer_brightness" as mode.

    // Set the "layer_color" argument to "texture_return.tint_call".
    mi::base::Handle<mi::neuraylib::IExpression_call> expression_layer_color(
      expression_factory->create_call( "texture_return.tint_call"));

    // Set the "mode" argument to "color_layer_brightness".
    mi::base::Handle<const mi::neuraylib::IType_enum> type_mode(
      type_factory->create_enum( "::base::color_layer_mode"));
    mi::Size index = type_mode->find_value( "color_layer_brightness");
    check_success( index != static_cast<mi::Size>( -1));
    mi::base::Handle<mi::neuraylib::IValue> value_mode(
      value_factory->create_enum( type_mode.get(), index));
    mi::base::Handle<mi::neuraylib::IExpression> expression_mode(
      expression_factory->create_constant( value_mode.get()));

    // Create the arguments list with both arguments above.
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "layer_color", expression_layer_color.get());
```

```
    arguments->add_expression( "mode", expression_mode.get());

   // Create a function call of the struct constructor for "mdl::base::color_layer" with
    // the just prepared arguments and store it as "color_layer0".
    mi::base::Handle<const mi::neuraylib::IFunction_definition> function_definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::base::color_layer(color,float,::base::color_layer_mode)"));
    mi::Sint32 result;
    mi::base::Handle<mi::neuraylib::IFunction_call> function_call(
      function_definition->create_function_call( arguments.get(), &result));
    check_success( result == 0);
    transaction->store( function_call.get(), "color_layer0");
  }
  {
    // Create a second color layer "color_layer1" with "layer_color" set to blue and
    // "color_layer_average" as mode. Store it as "color_layer1".

    // Set the "layer_color" argument to blue.
    mi::base::Handle<mi::neuraylib::IValue> value_layer_color(
      value_factory->create_color( 0.0f, 0.0f, 1.0f));
    mi::base::Handle<mi::neuraylib::IExpression_constant> expression_layer_color(
      expression_factory->create_constant( value_layer_color.get()));

    // Set the "mode" argument to "color_layer_average".
    mi::base::Handle<const mi::neuraylib::IType_enum> type_mode(
      type_factory->create_enum( "::base::color_layer_mode"));
    mi::Size index = type_mode->find_value( "color_layer_average");
    check_success( index != static_cast<mi::Size>( -1));
    mi::base::Handle<mi::neuraylib::IValue> value_mode(
      value_factory->create_enum( type_mode.get(), index));
    mi::base::Handle<mi::neuraylib::IExpression> expression_mode(
      expression_factory->create_constant( value_mode.get()));

    // Create the arguments list with both arguments above.
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "layer_color", expression_layer_color.get());
    arguments->add_expression( "mode", expression_mode.get());

   // Create a function call of the struct constructor for "mdl::base::color_layer" with
    // the just prepared arguments and store it as "color_layer1".
    mi::base::Handle<const mi::neuraylib::IFunction_definition> function_definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::base::color_layer(color,float,::base::color_layer_mode)"));
    mi::Sint32 result;
    mi::base::Handle<mi::neuraylib::IFunction_call> function_call(
      function_definition->create_function_call( arguments.get(), &result));
    check_success( result == 0);
    transaction->store( function_call.get(), "color_layer1");
  }
  {
   // Create an array of the two color layers "color_layer0" and "color_layer1" using the array
    // constructor. Store it as "color_layers".

   // Create the argument list with both arguments "color_layer0" and "color_layer1".
    mi::base::Handle<mi::neuraylib::IExpression> expression_color_layer0(
      expression_factory->create_call( "color_layer0"));
```

```
    mi::base::Handle<mi::neuraylib::IExpression> expression_color_layer1(
      expression_factory->create_call( "color_layer1"));
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "value0", expression_color_layer0.get());
    arguments->add_expression( "value1", expression_color_layer1.get());

   // Create a function call of the array constructor with the just prepared arguments and
    // store it as "color_layers".
    mi::base::Handle<const mi::neuraylib::IFunction_definition> function_definition(
      transaction->access<mi::neuraylib::IFunction_definition>( "mdl::T[](...)"));
    mi::Sint32 result;
    mi::base::Handle<mi::neuraylib::IFunction_call> function_call(
      function_definition->create_function_call( arguments.get(), &result));
    check_success( result == 0);
    transaction->store( function_call.get(), "color_layers");
  }
  {
    // Create a blend of the two color layers and red as base color. Store it as
    // "blended_color_layers".

    // Set the "layers" argument to "color_layers".
    mi::base::Handle<mi::neuraylib::IExpression> expression_layers(
      expression_factory->create_call( "color_layers"));

    // Set the "base" argument to red.
    mi::base::Handle<mi::neuraylib::IValue> value_base(
      value_factory->create_color( 1.0f, 0.0f, 0.0f));
    mi::base::Handle<mi::neuraylib::IExpression> expression_base(
      expression_factory->create_constant( value_base.get()));

    // Create the argument list with both arguments above.
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "layers", expression_layers.get());
    arguments->add_expression( "base", expression_base.get());

   // Create a function call from the definition "mdl::base::blend_color_layers" with the just
    // prepared arguments and store it as "blended_color_layers".
    mi::base::Handle<const mi::neuraylib::IFunction_definition> function_definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
     "mdl::base::blend_color_layers(::base::color_layer[P],color,::base::mono_mode)"));
    mi::Sint32 result;
    mi::base::Handle<mi::neuraylib::IFunction_call> function_call(
      function_definition->create_function_call( arguments.get(), &result));
    check_success( result == 0);
    transaction->store( function_call.get(), "blended_color_layers");
  }
  {
   // Prepare the arguments of the function call for "mdl::base::texture_return.tint": set the
    // "s" argument to the "file_texture_call" function call.
    mi::base::Handle<mi::neuraylib::IExpression> expression(
      expression_factory->create_call( "blended_color_layers"));
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "s", expression.get());
```

```
   // Create a function call from the definition "mdl::base::file_texture.tint" with the just
    // prepared arguments and store it as "texture_return.tint_call2".
    mi::base::Handle<const mi::neuraylib::IFunction_definition> function_definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::base::texture_return.tint(::base::texture_return)"));
    mi::Sint32 result;
    mi::base::Handle<mi::neuraylib::IFunction_call> function_call(
      function_definition->create_function_call( arguments.get(), &result));
    check_success( result == 0);
    transaction->store( function_call.get(), "texture_return.tint_call2");
  }
  {
    // Attach "texture_return.tint_call2" to the "tint" argument of "grey_material".
    mi::neuraylib::Argument_editor grey_material(
      transaction.get(), "grey_material", mdl_factory);
   check_success( grey_material.set_call( "tint", "texture_return.tint_call2") == 0);
  }

  render_and_export(
   neuray, transaction.get(), "file:example_mdl_struct_and_array_constructors.png");
  transaction->commit();
}

void create_variant(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::IScope* scope,
  mi::neuraylib::IMdl_factory* mdl_factory)
{
  // Create a transaction, MDL value and expression factory.
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IValue_factory> value_factory(
    mdl_factory->create_value_factory( transaction.get()));
  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction.get()));
  {
    // Prepare new defaults as clone of the current arguments of "grey_material".
    mi::base::Handle<const mi::neuraylib::IFunction_call> grey_material(
      transaction->access<mi::neuraylib::IFunction_call>( "grey_material"));
    mi::base::Handle<const mi::neuraylib::IExpression_list> arguments(
      grey_material->get_arguments());
    mi::base::Handle<mi::neuraylib::IExpression_list> defaults(
      expression_factory->clone( arguments.get()));

    // Create an ::anno::description annotation.
    mi::base::Handle<mi::neuraylib::IValue> anno_arg_value(
     value_factory->create_string( "a textured variant of ::main::diffuse_material"));
    mi::base::Handle<mi::neuraylib::IExpression> anno_arg_expression(
      expression_factory->create_constant( anno_arg_value.get()));
    mi::base::Handle<mi::neuraylib::IExpression_list> anno_args(
      expression_factory->create_expression_list());
    anno_args->add_expression( "description", anno_arg_expression.get());
    mi::base::Handle<mi::neuraylib::IAnnotation> anno(
    expression_factory->create_annotation( "::anno::description(string)", anno_args.get()));
    mi::base::Handle<mi::neuraylib::IAnnotation_block> anno_block(
      expression_factory->create_annotation_block());
```

```
    anno_block->add_annotation( anno.get());

    // Create the module bulder.
    mi::base::Handle<mi::neuraylib::IMdl_execution_context> context(
      mdl_factory->create_execution_context());
    mi::base::Handle<mi::neuraylib::IMdl_module_builder> module_builder(
      mdl_factory->create_module_builder(
        transaction.get(),
        "mdl::variants",
        mi::neuraylib::MDL_VERSION_1_0,
        mi::neuraylib::MDL_VERSION_LATEST,
        context.get()));
    check_success( module_builder);

    // Create the variant.
    check_success( module_builder->add_variant(
      "textured_material",
      "mdl::main::diffuse_material(color)",
      defaults.get(),
      anno_block.get(),
      /*return_annotations*/ 0,
      /*is_exported*/ true,
      context.get()) == 0);
  }
  {
    // Instantiate the variant.
   mi::base::Handle<const mi::neuraylib::IFunction_definition> textured_material_def(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::variants::textured_material(color)"));
    mi::Sint32 result = 0;
    mi::base::Handle<mi::neuraylib::IFunction_call> textured_material(
      textured_material_def->create_function_call( 0, &result));
    check_success( result == 0);
    transaction->store( textured_material.get(), "textured_material");

   // Attach the instantiated variant to the scene element "ground_instance", thereby
    // replacing the existing material instance "grey_material" it was created from.
    mi::base::Handle<mi::neuraylib::IInstance> instance(
        transaction->edit<mi::neuraylib::IInstance>( "ground_instance"));
   mi::base::Handle<mi::IArray> material( instance->edit_attribute<mi::IArray>( "material"));
    check_success(
    mi::set_value( material.get(), static_cast<mi::Size>( 0), "textured_material") == 0);
  }
  {
    // Export the variant.
   mi::base::Handle<mi::IArray> elements( transaction->create<mi::IArray>( "String[1]"));
    mi::base::Handle<mi::IString> element(
      elements->get_element<mi::IString>( static_cast<mi::Size>( 0)));
    element->set_c_str( "mdl::variants");
    mi::base::Handle<mi::neuraylib::IExport_api> export_api(
      neuray->get_api_component<mi::neuraylib::IExport_api>());
    mi::base::Handle<const mi::neuraylib::IExport_result> export_result(
    export_api->export_elements( transaction.get(), "file:variants.mdl", elements.get()));
  }

  render_and_export( neuray, transaction.get(), "file:example_mdl_variant.png");
  transaction->commit();
```

```
}

void distill(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::IScope* scope,
  mi::neuraylib::IMdl_factory* mdl_factory)
{
  // Create a transaction, MDL value, type and expression factory.
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction.get()));
  mi::base::Handle<mi::neuraylib::IValue_factory> value_factory(
    mdl_factory->create_value_factory( transaction.get()));
  mi::base::Handle<mi::neuraylib::IType_factory> type_factory(
    mdl_factory->create_type_factory( transaction.get()));

  {
    // Access and compile material instance
    mi::base::Handle<const mi::neuraylib::IMaterial_instance> grey_material(
      transaction->access<mi::neuraylib::IMaterial_instance>( "grey_material"));
    check_success(grey_material.is_valid_interface());

    mi::Uint32 flags = mi::neuraylib::IMaterial_instance::DEFAULT_OPTIONS;
    mi::base::Handle<const mi::neuraylib::ICompiled_material> comiled_material(
      grey_material->create_compiled_material(flags));
    check_success(comiled_material.is_valid_interface());

    // Use Distiller API to distill material to diffuse target
    mi::base::Handle<mi::neuraylib::IMdl_distiller_api> distiller_api(
      neuray->get_api_component<mi::neuraylib::IMdl_distiller_api>());
    check_success(distiller_api.is_valid_interface());

    mi::base::Handle<const mi::neuraylib::ICompiled_material> distilled_material(
      distiller_api->distill_material(comiled_material.get(), "diffuse"));
    check_success(distilled_material.is_valid_interface());

    // Bake a path from the distilled material and apply the
    // baking result to another material instance

    mi::base::Handle<const mi::neuraylib::IBaker> baker(distiller_api->create_baker(
      distilled_material.get(),
      "surface.scattering.tint"));
    check_success(baker.is_valid_interface());

    // Setup target material arguments
    mi::base::Handle<mi::neuraylib::IExpression_list> target_material_arguments(
      expression_factory->create_expression_list());

    mi::base::Handle<mi::neuraylib::IValue_bool> use_tex(
      value_factory->create_bool(!baker->is_uniform()));
    mi::base::Handle<mi::neuraylib::IExpression_constant> use_tex_expr(
      expression_factory->create_constant(use_tex.get()));
   target_material_arguments->add_expression("tint_use_texture", use_tex_expr.get());
```

```
    // Bake the path
    if(baker->is_uniform())
    {
      // Bake constant
      mi::base::Handle<mi::IColor> constant_color(
        transaction->create<mi::IColor>());
      check_success(baker->bake_constant(constant_color.get()) == 0);

      // Setup color argument for target material
      mi::Color c(0.0f, 0.0f, 0.0f);
      mi::get_value(constant_color.get(), c);
      mi::base::Handle<mi::neuraylib::IValue_color> color(
        value_factory->create_color(c.r, c.g, c.b));
      mi::base::Handle<mi::neuraylib::IExpression_constant> color_expr(
        expression_factory->create_constant(color.get()));

      target_material_arguments->add_expression("tint_color", color_expr.get());
    }
    else
    {
      // Create a canvas for storing the baked texture
      mi::base::Handle<mi::neuraylib::IImage_api> image_api(
        neuray->get_api_component<mi::neuraylib::IImage_api>());
      check_success(image_api.is_valid_interface());

      mi::base::Handle<mi::neuraylib::ICanvas> canvas(
        image_api->create_canvas("Rgb_fp", 1024, 1024));
      check_success(canvas.is_valid_interface());

      // Bake the texture
      check_success(baker->bake_texture(canvas.get(), 4) == 0);

      // Write the image to disk (for illustration purposes)
      mi::base::Handle<mi::neuraylib::IExport_api> export_api(
        neuray->get_api_component<mi::neuraylib::IExport_api>());
      check_success( export_api.is_valid_interface());
      export_api->export_canvas( "file:bake_result.png", canvas.get());

      // Store the baked texture into a database texture
      mi::base::Handle<mi::neuraylib::IImage> image(
        transaction->create<mi::neuraylib::IImage>( "Image"));
      check_success( image->set_from_canvas(canvas.get()));
      transaction->store( image.get(), "baked_image");
      mi::base::Handle<mi::neuraylib::ITexture> texture(
        transaction->create<mi::neuraylib::ITexture>( "Texture"));
      texture->set_image( "baked_image");
      transaction->store( texture.get(), "baked_texture");

      // Setup texture argument for target material
      mi::base::Handle<const mi::neuraylib::IType_texture> tex_type(
        type_factory->create_texture(mi::neuraylib::IType_texture::TS_2D));
      mi::base::Handle<mi::neuraylib::IValue_texture> tex(
        value_factory->create_texture(tex_type.get(), "baked_texture"));
      mi::base::Handle<mi::neuraylib::IExpression_constant> tex_ref(
        expression_factory->create_constant(tex.get()));

      target_material_arguments->add_expression("tint_texture", tex_ref.get());
```

```
    }

    // Create target material instance
    mi::neuraylib::Definition_wrapper target_material_def(
      transaction.get(),
      "mdl::main::textured_diffuse_material(texture_2d,bool,color)",
      mdl_factory);
    check_success(target_material_def.is_valid());

    mi::base::Handle<mi::neuraylib::IScene_element> target_material(
      target_material_def.create_instance(target_material_arguments.get()));
    transaction->store(target_material.get(), "target_material");

   // Attach the newly created material instance to the scene element "cube_instance"
    mi::base::Handle<mi::neuraylib::IInstance> instance(
      transaction->edit<mi::neuraylib::IInstance>( "cube_instance"));
   mi::base::Handle<mi::IArray> material( instance->edit_attribute<mi::IArray>( "material"));
    check_success(
    mi::set_value( material.get(), static_cast<mi::Size>( 0), "target_material") == 0);
  }

  render_and_export( neuray, transaction.get(), "file:example_mdl_distilling.png");
  transaction->commit();
}

void rendering( mi::neuraylib::INeuray* neuray)
{
  // Get the database and the global scope of the database.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());

  // Import the scene and render it.
  import_and_render_original_scene( neuray, scope.get());

 // Perform various MDL-related modifications, render the scene, and export the image to disk.
  mi::base::Handle<mi::neuraylib::IMdl_factory> mdl_factory(
    neuray->get_api_component<mi::neuraylib::IMdl_factory>());
  modify_material_instance( neuray, scope.get(), mdl_factory.get());
  create_new_material_instance( neuray, scope.get(), mdl_factory.get());
  attach_function_call( neuray, scope.get(), mdl_factory.get());
  use_struct_and_array_constructors( neuray, scope.get(), mdl_factory.get());
  create_variant( neuray, scope.get(), mdl_factory.get());
  distill( neuray, scope.get(), mdl_factory.get());
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_mdl <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];
```

```
  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.10 example_networking.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include <mi/neuraylib.h>

#include "example_shared.h"

class Host_callback : public mi::base::Interface_implement<mi::neuraylib::IHost_callback>
{
public:
  void connection_callback( mi::Uint32 host_id, bool flag)
  {
    if( flag) {
      fprintf( stderr,
       "The connection to the cluster was established. Own host id is %u.\n", host_id);
      m_own_host_id = host_id;
    }
    else
      fprintf( stderr, "The connection to the cluster was lost.\n");
  }
  void membership_callback( mi::Uint32 host_id, bool flag)
  {
    if( flag)
      fprintf( stderr, "Host %u joined the cluster.\n", host_id);
    else
      fprintf( stderr, "Host %u left the cluster.\n", host_id);
  }
 void property_callback( mi::Uint32 host_id, const mi::neuraylib::IHost_properties* properties)
  {
    fprintf( stderr, "Host %u communicated its properties:\n", host_id);
   mi::base::Handle<const mi::IString> value( properties->get_property( "application_name"));
    if( value.is_valid_interface())
      fprintf( stderr, "    application_name: %s\n", value->get_c_str());
  }
  void synchronizer_callback( mi::Uint32 host_id)
  {
    fprintf( stderr, "The synchronizer is now host %u", host_id);
    if( m_own_host_id == host_id)
      fprintf( stderr, " (this host)");
    fprintf( stderr, ".\n");
  }
  void database_status_callback( const char* status)
  {
    fprintf( stderr, "The database reports its status as \"%s\".\n", status);
  }

private:
  mi::Uint32 m_own_host_id;
};

void configuration(
```

```
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::IHost_callback* host_callback,
  int argc,
  char* argv[])
{
  mi::base::Handle<mi::neuraylib::INetwork_configuration> network_configuration(
    neuray->get_api_component<mi::neuraylib::INetwork_configuration>());

  // Register the callback handler
  network_configuration->register_host_callback( host_callback);

  // Set networking mode
  const char* mode = argv[1];
  if( strcmp( mode, ".") == 0) {
    ;
  } else if( strcmp( mode, "OFF") == 0) {
    check_success( network_configuration->set_mode(
      mi::neuraylib::INetwork_configuration::MODE_OFF) == 0);
  } else if( strcmp( mode, "TCP") == 0) {
    check_success( network_configuration->set_mode(
      mi::neuraylib::INetwork_configuration::MODE_TCP) == 0);
  } else if( strcmp( mode, "UDP") == 0) {
    check_success( network_configuration->set_mode(
      mi::neuraylib::INetwork_configuration::MODE_UDP) == 0);
  } else if( strcmp( mode, "TCP_WITH_DISCOVERY") == 0) {
    check_success( network_configuration->set_mode(
      mi::neuraylib::INetwork_configuration::MODE_TCP_WITH_DISCOVERY) == 0);
  } else {
    check_success( false);
  }

  // Set the multicast address
  if( strcmp( argv[2], ".") != 0)
    check_success( network_configuration->set_multicast_address( argv[2]) == 0);

  // Set the cluster interface
  if( strcmp( argv[3], ".") != 0)
    check_success( network_configuration->set_cluster_interface( argv[3]) == 0);

  // Set the discovery address
  if( strcmp( argv[4], ".") != 0)
    check_success( network_configuration->set_discovery_address( argv[4]) == 0);

  // Add configured hosts
  for( int i = 5; i < argc; ++i)
    check_success( network_configuration->add_configured_host( argv[i]) == 0);

  // Set a host property
  mi::base::Handle<mi::neuraylib::IGeneral_configuration> general_configuration(
    neuray->get_api_component<mi::neuraylib::IGeneral_configuration>());
 general_configuration->set_host_property( "application_name", "example_networking");

 // Load the Iray Photoreal plugin. This is only relevant when the example is used in conjunction
  // with the node manager example.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> plugin_configuration(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
  check_success( plugin_configuration->load_plugin_library(
```

```
    "libiray" MI_BASE_DLL_FILE_EXT) == 0);
}

void print_configuration( mi::neuraylib::INeuray* neuray)
{
  mi::base::Handle<mi::neuraylib::INetwork_configuration> network_configuration(
    neuray->get_api_component<mi::neuraylib::INetwork_configuration>());

  // Print networking mode
 mi::neuraylib::INetwork_configuration::Mode mode = network_configuration->get_mode();
  switch( mode) {
    case mi::neuraylib::INetwork_configuration::MODE_OFF:
      fprintf( stderr, "mode: OFF\n");
      break;
    case mi::neuraylib::INetwork_configuration::MODE_TCP:
      fprintf( stderr, "mode: TCP\n");
      break;
    case mi::neuraylib::INetwork_configuration::MODE_UDP:
      fprintf( stderr, "mode: UDP\n");
      break;
    case mi::neuraylib::INetwork_configuration::MODE_TCP_WITH_DISCOVERY:
      fprintf( stderr, "mode: TCP_WITH_DISCOVERY\n");
      break;
    default:
      fprintf( stderr, "mode: error\n");
      break;
  }

  // Print the multicast address
 mi::base::Handle<const mi::IString> string( network_configuration->get_multicast_address());
  fprintf( stderr, "multicast address: %s\n", string->get_c_str());

  // Print the cluster interface
  string = network_configuration->get_cluster_interface();
  fprintf( stderr, "cluster interface: %s\n", string->get_c_str());

  // Print the discovery address
  string = network_configuration->get_discovery_address();
  fprintf( stderr, "discovery address: %s\n", string->get_c_str());

  // Print the configured hosts
  fprintf( stderr, "configured hosts: ");
 for( mi::Uint32 i = 0; i < network_configuration->get_number_of_configured_hosts(); ++i) {
    string = network_configuration->get_configured_host( i);
    fprintf( stderr, "%s ", string->get_c_str());
  }
  fprintf( stderr, "\n");
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc < 5) {
   std::cerr << "Usage: example_networking <mode> <multicast_address> <cluster_interface>\\\n"
        << "         <discovery_address> [<host1> <host2> ... <hostN>]" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
```

```
  }

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Create callback handler
 mi::base::Handle<mi::neuraylib::IHost_callback> host_callback( new Host_callback());

  // Configure the neuray library
  configuration( neuray.get(), host_callback.get(), argc, argv);

  // Print the configuration
  print_configuration( neuray.get());

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Wait for other hosts to join/leave the cluster. The Host_callback objects prints
  // messages of joining or leaving hosts.
  sleep_seconds( 30);

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);

  // Unregister the callback handler again
  mi::base::Handle<mi::neuraylib::INetwork_configuration> network_configuration(
    neuray->get_api_component<mi::neuraylib::INetwork_configuration>());
  network_configuration->unregister_host_callback( host_callback.get());
  network_configuration = 0;
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.11 example_on_demand_mesh.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>
#include <vector>

class Cube : public mi::base::Interface_implement<mi::neuraylib::ISimple_mesh>
{
public:
  // Constructor. Sets up some of the member arrays of this class from a different representation.
   Cube();

  // Methods required by the mi::neuraylib::ISimple_mesh interface. In this implementation they
  // simply return pointers to member arrays of the class (or 0 for not-present optional data
   // arrays).
   mi::Uint32 data_size() const { return m_data_size; }
   const mi::Float32_3_struct* get_points() const { return &m_data_points[0]; }
   const mi::Float32_3_struct* get_normals() const { return &m_data_normals[0]; }
  mi::Uint32 get_texture_dimension( mi::Uint32 /*texture_space_id*/) const { return 0; }
  const mi::Float32* get_texture_coordinates( mi::Uint32 /*texture_space_id*/) const { return 0; }
   mi::Uint32 get_userdata_dimension(mi::Uint32 /*userdata_id*/) const { return 0; }
   const mi::Float32* get_userdata(mi::Uint32 /*userdata_id*/) const { return 0; }
   const char* get_userdata_name(mi::Uint32 /*userdata_id*/) const { return 0; }
   const mi::Float32_3_struct* get_derivatives() const { return 0; }
   mi::Uint32 get_motion_vector_count() const { return 0; }
   const mi::Float32_3_struct* get_motion_vectors() const { return 0; }
   mi::Uint32 triangles_size() const { return m_triangles_size; }
   const mi::Uint32_3_struct* get_triangles() const { return &m_triangles[0]; }
   bool has_unique_material() const { return true; }
   const mi::Uint32* get_material_indices() const { return 0; }

private:
   // These arrays hold the actual data used during runtime.
   // The first two are set up in the constructor from the arrays below.
   static const mi::Uint32 m_data_size = 24;
   static const mi::Uint32 m_triangles_size = 12;
   mi::Float32_3 m_data_points[m_data_size];
   mi::Float32_3 m_data_normals[m_data_size];
   static mi::Uint32_3 m_triangles[m_triangles_size];

   // These arrays hold the original data using a multi-index format.
   // They are used in the constructor to set up some of the arrays above.
   static const mi::Uint32 m_n_points = 8;
   static const mi::Uint32 m_n_normals = 6;
   static mi::Float32_3 m_points[m_n_points];
   static mi::Float32_3 m_normals[m_n_normals];
   static mi::Uint32 m_point_indices[m_data_size];
   static mi::Uint32 m_normal_indices[m_data_size];
```

```
};

mi::Float32_3 Cube::m_points[m_n_points] = {
  mi::Float32_3( -0.5, -0.5, -0.5 ),
  mi::Float32_3(  0.5, -0.5, -0.5 ),
  mi::Float32_3( -0.5,  0.5, -0.5 ),
  mi::Float32_3(  0.5,  0.5, -0.5 ),
  mi::Float32_3( -0.5, -0.5,  0.5 ),
  mi::Float32_3(  0.5, -0.5,  0.5 ),
  mi::Float32_3( -0.5,  0.5,  0.5 ),
  mi::Float32_3(  0.5,  0.5,  0.5 )
};

mi::Float32_3 Cube::m_normals[m_n_normals] = {
  mi::Float32_3(  0,  0, -1 ),
  mi::Float32_3(  0, -1,  0 ),
  mi::Float32_3( -1,  0,  0 ),
  mi::Float32_3( +1,  0,  0 ),
  mi::Float32_3(  0, +1,  0 ),
  mi::Float32_3(  0,  0, +1 )
};

mi::Uint32 Cube::m_point_indices[m_data_size] = {
  0, 2, 3, 1, 0, 1, 5, 4, 0, 4, 6, 2, 1, 3, 7, 5, 2, 6, 7, 3, 4, 5, 7, 6 };

mi::Uint32 Cube::m_normal_indices[m_data_size] = {
  0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5 };

mi::Uint32_3 Cube::m_triangles[m_triangles_size] = {
  mi::Uint32_3(  0,  1,  2 ),
  mi::Uint32_3(  0,  2,  3 ),
  mi::Uint32_3(  4,  5,  6 ),
  mi::Uint32_3(  4,  6,  7 ),
  mi::Uint32_3(  8,  9, 10 ),
  mi::Uint32_3(  8, 10, 11 ),
  mi::Uint32_3( 12, 13, 14 ),
  mi::Uint32_3( 12, 14, 15 ),
  mi::Uint32_3( 16, 17, 18 ),
  mi::Uint32_3( 16, 18, 19 ),
  mi::Uint32_3( 20, 21, 22 ),
  mi::Uint32_3( 20, 22, 23 )
};

Cube::Cube()
{
  for( mi::Size i = 0; i < m_data_size; ++i) {
    m_data_points[i] = m_points[m_point_indices[i]];
    m_data_normals[i] = m_normals[m_normal_indices[i]];
  }
}

class Cube_callback : public mi::base::Interface_implement<mi::neuraylib::IOn_demand_mesh_callback>
{
  const mi::neuraylib::ISimple_mesh* call() const { return new Cube(); }
};

void setup_scene( mi::neuraylib::ITransaction* transaction)
```

```
{
  mi::base::Handle<mi::neuraylib::IOn_demand_mesh> mesh(
    transaction->create<mi::neuraylib::IOn_demand_mesh>( "On_demand_mesh"));

  // Set the callback object that returns an instance of Cube.
 mi::base::Handle<mi::neuraylib::IOn_demand_mesh_callback> callback( new Cube_callback());
  mesh->set_callback( callback.get());
  callback = 0;

  // Set the remaining fields on the on-demand mesh.
  mesh->set_bbox( mi::Bbox3( -0.5, -0.5, -0.5, 0.5, 0.5, 0.5));
  mesh->set_maximum_displacement( 0);

  // Set the visible attribute and the material.
  mi::base::Handle<mi::IBoolean> visible(
    mesh->create_attribute<mi::IBoolean>( "visible", "Boolean"));
  visible->set_value( true);
  visible = 0;
 mi::base::Handle<mi::IRef> material( mesh->create_attribute<mi::IRef>( "material", "Ref"));
  check_success( material->set_reference( "yellow_material") == 0);
  material = 0;

  // Store the on-demand mesh.
  check_success( transaction->store( mesh.get(), "on_demand_mesh") == 0);
  mesh = 0;

  // Change the instance of the yellow cube to point to the on-demand mesh instead.
  mi::base::Handle<mi::neuraylib::IInstance> instance(
    transaction->edit<mi::neuraylib::IInstance>( "cube_instance"));
  check_success( instance->attach( "on_demand_mesh") == 0);
  instance = 0;
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
```

```
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Replace the triangle mesh by a equivalent on-demand mesh.
  setup_scene( transaction.get());

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_on_demand_mesh.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_on_demand_mesh <mdl_path>" << std::endl;
    keep_console_open();
```

```
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.12 example_particles.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>
#include <map>
#include <vector>

unsigned int lcg_a    = 134775815;
unsigned int lcg_c    = 123456;
unsigned int lcg_m    = 1 << 31;
unsigned int lcg_prev = 33478921;

unsigned int get_random_uint()
{
  mi::Uint64 val = (lcg_a * lcg_prev + lcg_c) % lcg_m;
  lcg_prev = static_cast<unsigned int>(val);
  return lcg_prev;
}

float get_random_float(const float& min, const float& max)
{
  return ((float(get_random_uint()) * (max - min)) / float(lcg_m)) + min;
}

mi::neuraylib::IParticles* create_particles_cuboid(mi::neuraylib::ITransaction* transaction)
{
  // Some constants to define the particles cuboid
  const mi::Float32_3 cuboid_center(-0.5f, 0.f, 0.75f);
  const mi::Float32   cuboid_x_length = 2.0f;
  const mi::Float32   cuboid_y_length = 0.5f;
  const mi::Float32   cuboid_z_length = 1.5f;
  const mi::Float32   min_particles_radius = 0.01f;
  const mi::Float32   max_particles_radius = 0.025f;
  const unsigned int  num_particles = 1000;

  // Constants derived from the ones above
  const mi::Float32 cuboid_min_x = cuboid_center.x - cuboid_x_length * 0.5f;
  const mi::Float32 cuboid_max_x = cuboid_center.x + cuboid_x_length * 0.5f;
  const mi::Float32 cuboid_min_y = cuboid_center.y - cuboid_y_length * 0.5f;
  const mi::Float32 cuboid_max_y = cuboid_center.y + cuboid_y_length * 0.5f;
  const mi::Float32 cuboid_min_z = cuboid_center.z - cuboid_z_length * 0.5f;
  const mi::Float32 cuboid_max_z = cuboid_center.z + cuboid_z_length * 0.5f;

 mi::neuraylib::IParticles* particles = transaction->create<mi::neuraylib::IParticles>("Particles"
  particles->set_type(mi::neuraylib::PARTICLE_TYPE_SPHERE);

 // In this example the particles will be added all at once, but they can also be added one by one
  std::vector<mi::Float32> particles_data;
```

```
  for (unsigned int n = 0; n < num_particles; ++n) {
    // Draw a random position inside the cuboid
    const float x = get_random_float(cuboid_min_x, cuboid_max_x);
    const float y = get_random_float(cuboid_min_y, cuboid_max_y);
    const float z = get_random_float(cuboid_min_z, cuboid_max_z);

    // Draw a random radius
   const float radius = get_random_float(min_particles_radius, max_particles_radius);

    particles_data.push_back(x);
    particles_data.push_back(y);
    particles_data.push_back(z);
    particles_data.push_back(radius);
  }

  // 'particles_data' now contains 4 times as many floats as there are particles
  particles->reserve(num_particles);
  particles->set_data(&particles_data.front(), num_particles);

  return particles;
}

void setup_scene( mi::neuraylib::ITransaction* transaction, const char* rootgroup)
{
  // Create the particles object
 mi::base::Handle<mi::neuraylib::IParticles> particles_object(create_particles_cuboid( transaction
  transaction->store(particles_object.get(), "particles_obj");

  // Create the instance for the particles object
  mi::base::Handle<mi::neuraylib::IInstance> instance(
    transaction->create<mi::neuraylib::IInstance>( "Instance"));
  instance->attach( "particles_obj");

  // Set the transformation matrix, the visible attribute, and the material
  mi::Float64_4_4 matrix( 1.0 );
  matrix.translate( 0.0, -0.4, 0.0);
  instance->set_matrix( matrix );

  mi::base::Handle<mi::IBoolean> visible(
    instance->create_attribute<mi::IBoolean>( "visible", "Boolean"));
  visible->set_value( true);

 mi::base::Handle<mi::IRef> material( instance->create_attribute<mi::IRef>( "material", "Ref"));
  material->set_reference( "red_material");

  transaction->store( instance.get(), "instance_particles");

  // Attach the instance to the root group
  mi::base::Handle<mi::neuraylib::IGroup> group(
    transaction->edit<mi::neuraylib::IGroup>( rootgroup));
  group->attach( "instance_particles");
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
```

```
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Add two triangle meshes to the scene
  setup_scene( transaction.get(), import_result->get_rootgroup());

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
```

```
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_particles.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_particles <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.13 example_plugins.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"

#include "imy_class.h"

void test_plugin( mi::neuraylib::INeuray* neuray)
{

  // Get the database, the global scope of the database, and create a transaction in the global
  // scope.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Create instance of My_class and call a method on it.
  mi::base::Handle<IMy_class> my_class( transaction->create<IMy_class>( "My_class"));
  check_success( my_class.is_valid_interface());
  my_class->set_foo( 42);

  // Store instance of My_class in the database and release the handle
  transaction->store( my_class.get(), "some_name");
  my_class = 0;

  // Get the instance of My_class from the database again
  my_class = transaction->edit<IMy_class>( "some_name");
  check_success( my_class.is_valid_interface());
  check_success( my_class->get_foo() == 42);
  my_class = 0;

  transaction->commit();
}

int main( int /*argc*/, char* /*argv*/[])
{
  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Load the plugin
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> plugin_configuration(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
  check_success( plugin_configuration->load_plugin_library(
    "./libplugin" MI_BASE_DLL_FILE_EXT) == 0);
  plugin_configuration = 0;
```

```
  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Interact with the loaded plugin
  test_plugin( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.14 example_polygon_mesh.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>

class Cylinder
{
public:
  Cylinder( mi::Uint32 n, mi::Float32 radius, mi::Float32 height)
    : m_n( n), m_radius( radius), m_height( height)
  {
    m_mesh_connectivity = new mi::Uint32[ 2*n + 4*n];
    m_normal_connectivity = new mi::Uint32[ 2*n + 4*n];
    m_offsets = new mi::Uint32[ n + 2 + 1];

    // offsets to the first vertex index of each polygon
    m_offsets[0] = 0;                                       // base (n vertices)
    m_offsets[1] = n;                                       // top (n vertices)
    for( mi::Uint32 i = 0; i <= n; ++i)
      m_offsets[2+i] = 2*n + 4*i;                         // sides (4 vertices each)

    // the mesh connectivity
    mi::Uint32 index = 0;
    for( mi::Uint32 i = 0; i < n; ++i)                  // base (first n even indices)
      m_mesh_connectivity[index++] = 2*i;
    for( mi::Uint32 i = 0; i < n; ++i)                   // top (first n odd indices)
      m_mesh_connectivity[index++] = 2*i+1;
   for( mi::Uint32 i = 0; i < n; ++i) {              // sides (four subsequent indices)
      m_mesh_connectivity[index++] =  2*i;
      m_mesh_connectivity[index++] = (2*i + 2) % (2*n);
      m_mesh_connectivity[index++] = (2*i + 3) % (2*n);
      m_mesh_connectivity[index++] = (2*i + 1) % (2*n);
    }

    // the custom connectivity for normals
    index = 0;
    for( mi::Uint32 i = 0; i < n; ++i)                  // base (one constant normal)
      m_normal_connectivity[index++] = 0;
    for( mi::Uint32 i = 0; i < n; ++i)                   // top (one constant normal)
      m_normal_connectivity[index++] = 1;
   for( mi::Uint32 i = 0; i < n; ++i) {              // sides (two normals each, shared
      m_normal_connectivity[index++] = 2 + i;           // with adjacent side face)
      m_normal_connectivity[index++] = 2 + (i+1) % n;
      m_normal_connectivity[index++] = 2 + (i+1) % n;
      m_normal_connectivity[index++] = 2 + i;
    }
  }
```

```
  ~Cylinder()
  {
    delete[] m_mesh_connectivity;
    delete[] m_normal_connectivity;
    delete[] m_offsets;
  }

  mi::Uint32 num_points() const { return m_n * 2; }

  mi::Uint32 num_normals() const { return m_n + 2; }

  mi::Uint32 num_polys() const { return m_n + 2; }

 mi::Uint32 polygon_size( mi::Uint32 p) const { return m_offsets[p+1] - m_offsets[p]; }

 mi::Uint32* polygon_indices( mi::Uint32 p) const { return &m_mesh_connectivity[m_offsets[p]]; }

 mi::Uint32* normal_indices( mi::Uint32 p) const { return &m_normal_connectivity[m_offsets[p]]; }

  mi::Float32_3 point( mi::Uint32 index)
  {
    mi::Uint32 i = index / 2;
    mi::Float32 angle = static_cast<mi::Float32>( 2.0f * MI_PI * i / m_n);

    if( index % 2 == 0)
      return mi::Float32_3(
        -m_height/2.0f, m_radius * std::sin( angle), m_radius * std::cos( angle));
    else
      return mi::Float32_3(
        m_height/2.0f, m_radius * std::sin( angle), m_radius * std::cos( angle));
  }

  mi::Float32_3 normal( mi::Uint32 index)
  {
    if( index == 0) return mi::Float32_3( -1.0f, 0.0f, 0.0f);
    if( index == 1) return mi::Float32_3(  1.0f, 0.0f, 0.0f);

    mi::Float32 angle = static_cast<mi::Float32>( 2.0f * MI_PI * (index-2) / m_n);
    return mi::Float32_3( 0.0f, std::sin( angle), std::cos( angle));
  }

private:
  mi::Uint32 m_n;
  mi::Float32 m_radius, m_height;
  mi::Uint32* m_mesh_connectivity;
  mi::Uint32* m_normal_connectivity;
  mi::Uint32* m_offsets;
};

mi::neuraylib::IPolygon_mesh* create_cylinder( mi::neuraylib::ITransaction* transaction,
  mi::Uint32 n, mi::Float32 radius, mi::Float32 height)
{
  Cylinder cylinder( n, radius, height);

  // Create an empty polygon mesh
  mi::neuraylib::IPolygon_mesh* mesh
    = transaction->create<mi::neuraylib::IPolygon_mesh>( "Polygon_mesh");
```

```
  check_success( mesh);

  // Create a cylinder (points and polygons)
  mesh->reserve_points( cylinder.num_points());
  for( mi::Uint32 i = 0; i < cylinder.num_points(); ++i)
    mesh->append_point( cylinder.point( i));
  for( mi::Uint32 i = 0; i < cylinder.num_polys(); ++i)
    mesh->add_polygon( cylinder.polygon_size( i));

  // Map vertices of the polygons to points
  mi::base::Handle<mi::neuraylib::IPolygon_connectivity> mesh_connectivity(
    mesh->edit_mesh_connectivity());
  for( mi::Uint32 i = 0; i < cylinder.num_polys(); ++i)
    mesh_connectivity->set_polygon_indices(
      mi::neuraylib::Polygon_handle( i), cylinder.polygon_indices( i));
  check_success( mesh->attach_mesh_connectivity( mesh_connectivity.get()) == 0);

  // Create a custom connectivity for normal vectors and map vertices to entries in the
  // attribute vector
  mi::base::Handle<mi::neuraylib::IPolygon_connectivity> normal_connectivity(
    mesh->create_connectivity());
  for( mi::Uint32 i = 0; i < cylinder.num_polys(); ++i)
    normal_connectivity->set_polygon_indices(
      mi::neuraylib::Polygon_handle( i), cylinder.normal_indices( i));

  // Create an attribute vector for the normals
  mi::base::Handle<mi::neuraylib::IAttribute_vector> normals(
    normal_connectivity->create_attribute_vector( mi::neuraylib::ATTR_NORMAL));
  for( mi::Uint32 i = 0; i < cylinder.num_normals(); ++i)
    normals->append_vector3( cylinder.normal( i));
  check_success( normals->is_valid_attribute());
  check_success( normal_connectivity->attach_attribute_vector( normals.get()) == 0);
  check_success( !normals->is_valid_attribute());

  check_success( mesh->attach_connectivity( normal_connectivity.get()) == 0);

  return mesh;
}

void setup_scene( mi::neuraylib::ITransaction* transaction, const char* rootgroup)
{
  mi::Float32 cylinder_radius = 0.6f;
  mi::Float32 cylinder_height = 1.2f;

  // Create the red cylinder
  mi::base::Handle<mi::neuraylib::IPolygon_mesh> mesh_red( create_cylinder(
    transaction, 23, cylinder_radius, cylinder_height));
  transaction->store( mesh_red.get(), "mesh_red");

  // Create the instance for the red cylinder
  mi::base::Handle<mi::neuraylib::IInstance> instance(
    transaction->create<mi::neuraylib::IInstance>( "Instance"));
  instance->attach( "mesh_red");

  // Set the transformation matrix, the visible attribute, and the material
  mi::Float64_4_4 matrix( 1.0);
  matrix.translate( -0.1, -cylinder_radius, 0.2);
```

```
  matrix.rotate( 0.0, -0.5 * MI_PI_2, 0.0);
  instance->set_matrix( matrix);

  mi::base::Handle<mi::IBoolean> visible(
    instance->create_attribute<mi::IBoolean>( "visible", "Boolean"));
  visible->set_value( true);

 mi::base::Handle<mi::IRef> material( instance->create_attribute<mi::IRef>( "material", "Ref"));
  material->set_reference( "red_material");

  transaction->store( instance.get(), "instance_red");

  // And attach the instance to the root group
  mi::base::Handle<mi::neuraylib::IGroup> group(
    transaction->edit<mi::neuraylib::IGroup>( rootgroup));
  group->attach( "instance_red");

  // Tessellate the polygon mesh of the red cylinder.
  mi::base::Handle<mi::neuraylib::ITessellator> tessellator(
    transaction->create<mi::neuraylib::ITessellator>( "Tessellator"));
  mi::base::Handle<const mi::neuraylib::IPolygon_mesh> c_mesh_red(
    transaction->access<mi::neuraylib::IPolygon_mesh>( "mesh_red"));
  mi::base::Handle<mi::neuraylib::ITriangle_mesh> m_mesh_blue(
    tessellator->run( c_mesh_red.get()));
  transaction->store( m_mesh_blue.get(), "mesh_blue");

  // Create the instance for the blue cylinder
  instance = transaction->create<mi::neuraylib::IInstance>( "Instance");
  instance->attach( "mesh_blue");

  // Set the transformation matrix, the visible attribute, and the material
  matrix = mi::Float64_4_4( 1.0);
  matrix.translate( -1.1, -cylinder_radius, -1.1);
  instance->set_matrix( matrix);

  visible = instance->create_attribute<mi::IBoolean>( "visible", "Boolean");
  visible->set_value( true);

 mi::base::Handle<mi::IRef> ref( instance->create_attribute<mi::IRef>( "material", "Ref"));
  ref->set_reference( "blue_material");

  transaction->store( instance.get(), "instance_blue");

  // And attach the instance to the root group
  group->attach( "instance_blue");
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
```

```
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
  check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray)
{
  // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Add a polygon mesh and a tessellated blue mesh to the scene
  setup_scene( transaction.get(), import_result->get_rootgroup());

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
  check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
```

```
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_polygon_mesh.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_polygon_mesh <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.15 example_progressive_rendering.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iomanip>
#include <iostream>
#include <sstream>

#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_advanced.h"

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray, const char* scene_file)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
 mi::base::Handle<const mi::IString> uri( import_api->convert_filename_to_uri( scene_file));
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), uri->get_c_str()));
  check_success( import_result->get_error_number() == 0);

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
```

```
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  scene = 0;

  // Render progressively at most 10 frames
  for( mi::Size i = 0; i < 10; ++i) {

    // Create the render target and render the i-th frame of the scene
    mi::base::Handle<mi::neuraylib::IRender_target> render_target(
      new Render_target( 512, 384));
   mi::Sint32 result = render_context->render( transaction.get(), render_target.get(), 0);
    check_success( result >= 0);

    // Write the image to disk
    mi::base::Handle<mi::neuraylib::IExport_api> export_api(
      neuray->get_api_component<mi::neuraylib::IExport_api>());
    check_success( export_api.is_valid_interface());
    mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
    std::ostringstream str;
   str << "example_progressive_rendering_" << std::setw( 2) << std::setfill( '0') << i
      << ".png";
    export_api->export_canvas( str.str().c_str(), canvas.get());

    // Leave render loop if the termination criteria have been met
    if( result > 0)
      break;
  }

  // Pick the cube.
  mi::base::Handle<mi::neuraylib::IPick_array> pick_array(
    render_context->pick( transaction.get(), mi::Float32_2( 128, 192)));
  std::cerr << "Picked objects:" << std::endl;
  for( mi::Uint32 i = 0; i < pick_array->get_length(); ++i) {
   mi::base::Handle<mi::neuraylib::IPick_result> pick_result( pick_array->get_pick_result( i));
    mi::Float64_3 hit_point = pick_result->get_world_point();
   std::cerr << "Object " << i << ": \"" << pick_result->get_picked_object_name() << "\", "
    << "hit point (" << hit_point.x << ", " << hit_point.y << ", " << hit_point.z << "), "
      << "path \"" << pick_result->get_path( 0) << "\"";
    mi::Uint32 path_length = pick_result->get_path_length();
    for( mi::Uint32 j = 1; j < path_length; ++j)
      std::cerr << ", \"" << pick_result->get_path( j) << "\"";
    std::cerr << std::endl;
  }

  transaction->commit();
}

int main( int argc, char* argv[])
{
```

```
  // Collect command line parameters
  if( argc != 3) {
   std::cerr << "Usage: example_progressive_rendering <scene_file> <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* scene_file = argv[1];
  const char* mdl_path   = argv[2];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get(), scene_file);

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.16 example_psd_exporter.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <cassert>
#include <cstdio>
#include <iostream>
#include <limits>

#include <mi/neuraylib.h>

#include "example_shared.h"

#ifndef MI_PLATFORM_WINDOWS
#include <arpa/inet.h>
#else
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN 1
#endif
#include <winsock2.h>
#endif


class Render_target : public mi::base::Interface_implement<mi::neuraylib::IRender_target>
{
public:
  // Constructor. Creates a render target with three canvases.
 Render_target( mi::Uint32 width, mi::Uint32 height, mi::neuraylib::IImage_api* image_api);

  // Implement the interface of mi::neuraylib::IRender_target.
  mi::Uint32 get_canvas_count() const;
  mi::neuraylib::Canvas_type get_canvas_type( mi::Uint32 index) const;
  const char* get_canvas_name( mi::Uint32 index) const;
  const mi::neuraylib::ICanvas_parameters* get_canvas_parameters( mi::Uint32) const;
  const mi::neuraylib::ICanvas* get_canvas( mi::Uint32 index) const;
  mi::neuraylib::ICanvas* get_canvas( mi::Uint32 index);

  // Normalizes the contents of the second and third canvas.
  void normalize();

private:
  // The three canvases of this render target.
  mi::base::Handle<mi::neuraylib::ICanvas> m_canvas[3];
};


FILE* g_file;

void write_int8( mi::Uint8 data)
{
  fwrite( &data, 1, 1, g_file);
}

void write_int16( mi::Uint16 data)
```

```
{
  data = htons( data);
  fwrite( &data, 2, 1, g_file);
}

void write_int32( mi::Uint32 data)
{
  data = htonl( data);
  fwrite( &data, 4, 1, g_file);
}

void write_buf( const void* p, mi::Size length)
{
  fwrite( p, length, 1, g_file);
}

void write_str( const char* s)
{
  write_buf( s, strlen( s));
}

void write_str_pascal( const char* s, mi::Size padding)
{
  mi::Size len = strlen( s);
  if( len > 255)
    len = 255;
  write_int8( static_cast<mi::Uint8>( len));
  write_buf( s, len);
  for( mi::Size i = (len % padding) + 1; i < padding; ++i)
    write_int8( 0);
}

void write_channel(
  const mi::Uint8* input,
  mi::Uint32 width,
  mi::Uint32 height,
  mi::Uint32 bytes_per_channel,
  mi::Uint32 stride)
{
  mi::Uint8* output = new mi::Uint8[width * height * bytes_per_channel];
  for( mi::Uint32 i = 0; i < width * height; ++i)
    for( mi::Uint32 b = 0; b < bytes_per_channel; ++b)
      output[i*bytes_per_channel + b] = input[i*stride + bytes_per_channel-1-b];
  write_buf( output, width * height * bytes_per_channel);
  delete[] output;
}

mi::Uint32 get_channel_count( const char* pixel_type)
{
  if( strcmp( pixel_type, "Float32<4>") == 0) return 4;
  if( strcmp( pixel_type, "Rgba"      ) == 0) return 4;
  if( strcmp( pixel_type, "Rgbea"     ) == 0) return 4;
  if( strcmp( pixel_type, "Rgba_16"   ) == 0) return 4;
  if( strcmp( pixel_type, "Color"     ) == 0) return 4;
  return 3;
}
```

```
mi::Uint32 get_bytes_per_channel( const char* pixel_type)
{
  if( strcmp( pixel_type, "Sint8"  ) == 0) return 1;
  if( strcmp( pixel_type, "Sint32" ) == 0) return 1;
  if( strcmp( pixel_type, "Rgb"    ) == 0) return 1;
  if( strcmp( pixel_type, "Rgba"   ) == 0) return 1;
  return 2;
}

const char* get_pixel_type( mi::Uint32 bytes_per_channel, mi::Uint32 channels)
{
  assert( channels == 3 || channels == 4);

  switch( bytes_per_channel) {
    case 1:  return channels == 3 ? "Rgb"    : "Rgba";
    case 2:  return channels == 3 ? "Rgb_16" : "Rgba_16";
    default: assert( false); return "Rgb";
  }
}

bool export_psd(
  const mi::IArray* canvases,
  const mi::IArray* names,
  const char* filename,
  mi::neuraylib::IImage_api* image_api)
{
  if( !canvases || !filename || !image_api)
    return false;

  // Check consistent array length
  if( names && names->get_length() != canvases->get_length())
    return false;

  // Compute maximum width, height, bytes per channel, and total number of layers
  mi::Uint32 total_width = 0;
  mi::Uint32 total_height = 0;
  mi::Uint32 total_layers = 0;
  mi::Uint32 bytes_per_channel = 0;
  for( mi::Uint32 i = 0; i < canvases->get_length(); ++i) {
    mi::base::Handle<const mi::neuraylib::ICanvas> canvas(
      canvases->get_element<mi::neuraylib::ICanvas>( i));
    if( !canvas.is_valid_interface())
      return false;
    total_width  = std::max( total_width,  canvas->get_resolution_x());
    total_height = std::max( total_height, canvas->get_resolution_y());
    total_layers += canvas->get_layers_size();
    bytes_per_channel = std::max( bytes_per_channel,
      get_bytes_per_channel( canvas->get_type()));
  }

  // Reject canvases too large for PSD files version 1
  if( total_width > 30000 || total_height > 30000)
    return false;

  // Reject if too many layers in total
  if( total_layers == 0 || total_layers > 56)
    return false;
```

```
  // Check names
  bool has_result = false;
  for( mi::Uint32 i = 0; names && i < names->get_length(); ++i) {
    mi::base::Handle<const mi::IString> name(
      names->get_element<mi::IString>( i));
    if( !name.is_valid_interface())
      return false;
    if( strcmp( name->get_c_str(), "result") == 0)
      has_result = true;
  }

  g_file = fopen( filename, "wb");
  if( !g_file)
    return false;

  // File header section

  write_str( "8BPS");                 // signature
  write_int16( 1);                    // version
  write_buf( "\0\0\0\0\0\0", 6);      // reserved
  write_int16( 3);                    // number of channels
  write_int32( total_height);         // height
  write_int32( total_width);          // width
  write_int16( static_cast<mi::Uint16>( 8*bytes_per_channel)); // bits per channel
  write_int16( 3);                    // color mode (RGB)

  write_int32( 0);                    // length of color mode data section
  write_int32( 0);                    // length of image resource section

  // Layer and mask information section

 write_int32( 0);              // length of layer and mask information section (dummy)
  long lamis_start = ftell( g_file);

  // Layer and mask information section: Layer info subsection

  write_int32( 0);                    // length of layer information section (dummy)
  long lis_start = ftell( g_file);

  write_int16( static_cast<mi::Uint16>( total_layers)); // layer count

  // Layer records

  for( mi::Size i = 0; i < canvases->get_length(); ++i) {

    mi::base::Handle<const mi::neuraylib::ICanvas> canvas(
      canvases->get_element<mi::neuraylib::ICanvas>( i));
    mi::Uint32 width = canvas->get_resolution_x();

    const char* name = "";
    if( names) {
      mi::base::Handle<const mi::IString> s( names->get_element<mi::IString>( i));
      name = s->get_c_str();
    }
    bool is_result = strcmp( name, "result") == 0;
```

```
    for( mi::Size j = 0; j < canvas->get_layers_size(); ++j) {

      mi::Uint32 channels = get_channel_count( canvas->get_type());

      write_int32( 0);                    // top
      write_int32( 0);                    // left
      write_int32( total_height);         // bottom
      write_int32( width);                // right
      write_int16( static_cast<mi::Uint16>( channels)); // number of channels
      mi::Uint32 channel_size = width * total_height * bytes_per_channel + 2;
      write_int16( 0);                    // channel ID red
      write_int32( channel_size);         // byte count red
      write_int16( 1);                    // channel ID green
      write_int32( channel_size);         // byte count green
      write_int16( 2);                    // channel ID blue
      write_int32( channel_size);         // byte count blue
      if( channels == 4) {
        write_int16( 0xFFFF);           // channel ID alpha
        write_int32( channel_size);     // byte count alpha
      }

      write_str( "8BIM");                 // blend mode signature
      write_str( "norm");                 // blend mode key
      write_int8( 255);                   // opacity
      write_int8( 0);                     // clipping
      mi::Uint8 flags = static_cast<mi::Uint8>( !has_result || is_result ? 0 : 2);
      write_int8( flags);                 // flags (invisible = 2)
      write_int8( 0);                     // filler
      write_int32( 0);                    // extra data length (dummy)
      long extra_data_start = ftell( g_file);
      write_int32( 0);                    // layer mask data length
      write_int32( 0);                    // layer blending ranges length
      write_str_pascal( name, 4);         // layer name

      // extra data length
      long extra_data_end = ftell( g_file);
      fseek( g_file, extra_data_start-4, SEEK_SET);
      write_int32( static_cast<mi::Uint32>( extra_data_end-extra_data_start));
      fseek( g_file, extra_data_end, SEEK_SET);
    }
  }

  // Channel image data

  for( mi::Size i = 0; i < canvases->get_length(); ++i) {

    mi::base::Handle<const mi::neuraylib::ICanvas> canvas(
      canvases->get_element<mi::neuraylib::ICanvas>( i));
    mi::Uint32 width  = canvas->get_resolution_x();
    mi::Uint32 height = canvas->get_resolution_y();

    for( mi::Uint32 j = 0; j < canvas->get_layers_size(); ++j) {

      mi::Uint32 channels = get_channel_count( canvas->get_type());
    mi::Uint8* buffer = new mi::Uint8[width * total_height * channels * bytes_per_channel];
      memset( buffer, 0, width * total_height * channels * bytes_per_channel);
     mi::Size offset = width * (total_height-height) * channels * bytes_per_channel;
```

```
    const char* pixel_type = get_pixel_type( bytes_per_channel, channels);
    image_api->read_raw_pixels(
      width, height, canvas.get(), 0, 0, j, buffer + offset, true, pixel_type);

    for( mi::Uint32 channel = 0; channel < channels; ++channel) {
      write_int16( 0);               // channel compression method (raw)
      mi::Uint8* start = buffer + channel * bytes_per_channel;
      mi::Uint32 stride = channels * bytes_per_channel;
      write_channel( start, width, total_height, bytes_per_channel, stride);
    }

    delete[] buffer;
  }
}

// length of layer information section
long lis_end = ftell( g_file);
fseek( g_file, lis_start-4, SEEK_SET);
write_int32( static_cast<mi::Uint32>( lis_end-lis_start));
fseek( g_file, lis_end, SEEK_SET);

// length of global layer mask info
write_int32( 0);

// length of layer and mask inform. section
long lamis_end = ftell( g_file);
fseek( g_file, lamis_start-4, SEEK_SET);
write_int32( static_cast<mi::Uint32>( lamis_end-lamis_start));
fseek( g_file, lamis_end, SEEK_SET);

// Image data section

mi::base::Handle<const mi::neuraylib::ICanvas> canvas(
  canvases->get_element<mi::neuraylib::ICanvas>( 0));
mi::Uint32 width  = canvas->get_resolution_x();
mi::Uint32 height = canvas->get_resolution_y();

for( mi::Uint32 j = 0; j < canvas->get_layers_size(); ++j) {

  mi::Uint32 channels = 3;
  mi::Uint8* buffer
    = new mi::Uint8[total_width * total_height * channels * bytes_per_channel];
  memset( buffer, 0, total_width * total_height * channels * bytes_per_channel);
 mi::Size offset = total_width * (total_height-height) * channels * bytes_per_channel;
  mi::Uint32 padding = (total_width - width) * channels * bytes_per_channel;
  const char* pixel_type = get_pixel_type( bytes_per_channel, channels);
  image_api->read_raw_pixels(
   width, height, canvas.get(), 0, 0, j, buffer + offset, true, pixel_type, padding);

  write_int16( 0);                    // compression method (raw)
  for( mi::Uint32 channel = 0; channel < channels; ++channel) {
    mi::Uint8* start = buffer + channel * bytes_per_channel;
    mi::Uint32 stride = channels * bytes_per_channel;
    write_channel( start, width, total_height, bytes_per_channel, stride);
  }

  delete[] buffer;
```

```
  }

  fclose( g_file);
  return true;
}

bool export_psd(
  Render_target* render_target,
  const char* filename,
  mi::neuraylib::IFactory* factory,
  mi::neuraylib::IImage_api* image_api)
{
  if( !render_target || !filename || !image_api)
    return false;

  mi::base::Handle<mi::IDynamic_array> canvases(
    factory->create<mi::IDynamic_array>( "Interface[]"));
  check_success( canvases.is_valid_interface());
  mi::base::Handle<mi::IDynamic_array> names(
    factory->create<mi::IDynamic_array>( "String[]"));
  check_success( names.is_valid_interface());
  for( mi::Uint32 i = 0; i < render_target->get_canvas_count(); ++i) {
    mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( i));
    canvases->push_back( canvas.get());
    mi::base::Handle<mi::IString> name( factory->create<mi::IString>( "String"));
    name->set_c_str( render_target->get_canvas_name( i));
    names->push_back( name.get());
  }

  return export_psd( canvases.get(), names.get(), filename, image_api);
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

  // Constructor. Creates a render target with three canvases.
 Render_target::Render_target( mi::Uint32 width, mi::Uint32 height, mi::neuraylib::IImage_api* imag
  {
    m_canvas[0] = image_api->create_canvas( "Color", width, height, 1);
    m_canvas[1] = image_api->create_canvas( "Float32<3>", width, height, 1);
    m_canvas[2] = image_api->create_canvas( "Float32", width, height, 1);
  }

  // Implement the interface of mi::neuraylib::IRender_target.
```

```
mi::Uint32 Render_target::get_canvas_count() const { return 3; }
mi::neuraylib::Canvas_type Render_target::get_canvas_type( mi::Uint32 index) const
{
  switch (index) {
    case 0:  return mi::neuraylib::TYPE_RESULT;
    case 1:  return mi::neuraylib::TYPE_NORMAL;
    case 2:  return mi::neuraylib::TYPE_DEPTH;
    default: return mi::neuraylib::TYPE_UNDEFINED;
  }
}
const char* Render_target::get_canvas_name( mi::Uint32 index) const
{
  switch (index) {
    case 0: return "result";
    case 1: return "normal";
    case 2: return "depth";
    default: return 0;
  }
}
const mi::neuraylib::ICanvas_parameters* Render_target::get_canvas_parameters( mi::Uint32) const
{ return 0; }
const mi::neuraylib::ICanvas* Render_target::get_canvas( mi::Uint32 index) const
{
  if( index >= 3)
    return 0;
  m_canvas[index]->retain();
  return m_canvas[index].get();
}
mi::neuraylib::ICanvas* Render_target::get_canvas( mi::Uint32 index)
{
  if( index >= 3)
    return 0;
  m_canvas[index]->retain();
  return m_canvas[index].get();
}

// Normalizes the contents of the second and third canvas.
void Render_target::normalize()
{

  // Map values in m_canvas[1] linearly from [-1,1] to [0,1].
  mi::base::Handle<mi::neuraylib::ITile> tile( m_canvas[1]->get_tile());
  mi::Size n_pixels = tile->get_resolution_x() * tile->get_resolution_y();
  mi::Float32_3* data1 = static_cast<mi::Float32_3*>( tile->get_data());
  for( mi::Size i = 0; i < n_pixels; ++i) {
    data1[i].x = mi::math::clamp( 0.5f*data1[i].x + 0.5f, 0.0f, 1.0f);
    data1[i].y = mi::math::clamp( 0.5f*data1[i].y + 0.5f, 0.0f, 1.0f);
    data1[i].z = mi::math::clamp( 0.5f*data1[i].z + 0.5f, 0.0f, 1.0f);
  }

  // Map values in m_canvas[2] (excluding huge values) linearly to [0,1].
  tile = m_canvas[2]->get_tile();
  n_pixels = tile->get_resolution_x() * tile->get_resolution_y();
  mi::Float32* data2 = static_cast<mi::Float32*>( tile->get_data());
  mi::Float32 min_value = std::numeric_limits<mi::Float32>::max();
  mi::Float32 max_value = std::numeric_limits<mi::Float32>::min();
  for( mi::Size i = 0; i < n_pixels; ++i) {
```

```
      if (data2[i] < min_value) min_value = data2[i];
      if (data2[i] > max_value && data2[i] < 1.0E38f) max_value = data2[i];
    }
    if( min_value == max_value)
      min_value = max_value-1;
    for( mi::Size i = 0; i < n_pixels; ++i)
      if( data2[i] < 1.0E38f)
        data2[i] = 1.0f - (data2[i] - min_value) / (max_value - min_value);
      else
        data2[i] = 0.0f;
  }

void rendering(
  mi::neuraylib::INeuray* neuray, const char* scene_file)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
 mi::base::Handle<const mi::IString> uri( import_api->convert_filename_to_uri( scene_file));
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), uri->get_c_str()));
  check_success( import_result->get_error_number() == 0);

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
 mi::base::Handle<Render_target> render_target( new Render_target( 512, 384, image_api.get()));
```

```
  check_success(
    render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk (entire render target as one PSD file)
  render_target->normalize();
  mi::base::Handle<mi::neuraylib::IFactory> factory(
    neuray->get_api_component<mi::neuraylib::IFactory>());
  check_success( export_psd(
   render_target.get(), "example_psd_exporter.psd", factory.get(), image_api.get()));

  // Write the image to disk (individual canvases as PNG files)
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas0( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_psd_exporter_0.png", canvas0.get());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas1( render_target->get_canvas( 1));
  export_api->export_canvas( "file:example_psd_exporter_1.png", canvas1.get());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas2( render_target->get_canvas( 2));
  export_api->export_canvas( "file:example_psd_exporter_2.png", canvas2.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 3) {
    std::cerr << "Usage: example_psd_exporter <scene_file> <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* scene_file = argv[1];
  const char* mdl_path   = argv[2];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get(), scene_file);

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
```

```
  return EXIT_SUCCESS;
}
```

20.17 example_render_target_advanced.h

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#ifndef EXAMPLE_RENDER_TARGET_ADVANCED_H
#define EXAMPLE_RENDER_TARGET_ADVANCED_H

#include <mi/neuraylib.h>

class Tile : public mi::base::Interface_implement<mi::neuraylib::ITile>
{
public:
  // Constructor.
  //
  // Creates a tile of the given width and height with pixel type "Color".
  Tile( mi::Uint32 width, mi::Uint32 height)
  {
    m_width = width;
    m_height = height;
    m_data = new mi::Float32[m_width * m_height * 4];
  }

  // Destructor
  ~Tile() { delete[] m_data; }

  // Implement the interface of mi::neuraylib::ITile
  void set_pixel( mi::Uint32 x_offset, mi::Uint32 y_offset, const mi::Float32* floats)
  {
    mi::Float32* position = &m_data[(x_offset + y_offset * m_width) * 4];
    position[0] = floats[0];
    position[1] = floats[1];
    position[2] = floats[2];
    position[3] = floats[3];
  }
  void get_pixel( mi::Uint32 x_offset, mi::Uint32 y_offset, mi::Float32* floats) const
  {
    mi::Float32* position = &m_data[(x_offset + y_offset * m_width) * 4];
    floats[0] = position[0];
    floats[1] = position[1];
    floats[2] = position[2];
    floats[3] = position[3];
  }
  const char* get_type() const { return "Color"; }
  mi::Uint32 get_resolution_x() const { return m_width; }
  mi::Uint32 get_resolution_y() const { return m_height; }
  const void* get_data() const { return m_data; }
  void* get_data() { return m_data; }

private:
  // Width of the tile
  mi::Uint32 m_width;
  // Height of the tile
  mi::Uint32 m_height;
  // The data of this tile, 32 bytes per pixel
```

```
  mi::Float32* m_data;
};

class Canvas : public mi::base::Interface_implement<mi::neuraylib::ICanvas>
{
public:
  // Constructor.
  //
  // Creates a canvas with a single tile of the given width and height.
  Canvas( mi::Uint32 width, mi::Uint32 height)
  {
    m_width = width;
    m_height = height;
    m_gamma = 1.0f;
    m_tile = new Tile( width, height);
  }

  // Implement the interface of mi::neuraylib::ICanvas
  mi::Uint32 get_resolution_x() const { return m_width; }
  mi::Uint32 get_resolution_y() const { return m_height; }
  mi::Uint32 get_layers_size() const { return 1; }
  const mi::neuraylib::ITile* get_tile(mi::Uint32 = 0) const
  {
    m_tile->retain();
    return m_tile.get();
  }
  mi::neuraylib::ITile* get_tile(mi::Uint32 = 0)
  {
    m_tile->retain();
    return m_tile.get();
  }
  const char* get_type() const { return "Color"; }
  mi::Float32 get_gamma() const { return m_gamma; }
  void set_gamma( mi::Float32 gamma) { m_gamma = gamma; }

private:
  // Width of the canvas
  mi::Uint32 m_width;
  // Height of the canvas
  mi::Uint32 m_height;
  // Gamma value of the canvas
  mi::Float32 m_gamma;
  // The only tile of this canvas
  mi::base::Handle<mi::neuraylib::ITile> m_tile;
};

class Render_target : public mi::base::Interface_implement<mi::neuraylib::IRender_target>
{
public:
  // Constructor.
  //
  // Creates a render target with a single canvas of the given width and height.
  // This variant uses custom implementations for canvases and tiles that are defined above
  // by the classes Tile and Canvas. In these implementations the pixel type is fixed to "Color"
  // for simplicity.
  Render_target( mi::Uint32 width, mi::Uint32 height)
  {
```

```
    m_canvas = new Canvas( width, height);
  }

  // Implement the interface of mi::neuraylib::IRender_target
  mi::Uint32 get_canvas_count() const { return 1; }
  mi::neuraylib::Canvas_type get_canvas_type( mi::Uint32 index) const
  { return index == 0 ? mi::neuraylib::TYPE_RESULT : mi::neuraylib::TYPE_UNDEFINED; }
  const mi::neuraylib::ICanvas_parameters* get_canvas_parameters( mi::Uint32) const
  { return 0; }
  const mi::neuraylib::ICanvas* get_canvas( mi::Uint32 index) const
  {
    if( index > 0)
      return 0;
    m_canvas->retain();
    return m_canvas.get();
  }
  mi::neuraylib::ICanvas* get_canvas( mi::Uint32 index)
  {
    if( index > 0)
      return 0;
    m_canvas->retain();
    return m_canvas.get();
  }

private:
  // The only canvas of this render target
  mi::base::Handle<mi::neuraylib::ICanvas> m_canvas;
};

#endif // MI_EXAMPLE_RENDER_TARGET_ADVANCED_H
```

20.18 example_rendering.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering(
  mi::neuraylib::INeuray* neuray, const char* scene_file)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
 mi::base::Handle<const mi::IString> uri( import_api->convert_filename_to_uri( scene_file));
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), uri->get_c_str()));
  check_success( import_result->get_error_number() == 0);

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
```

```
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_rendering.png", canvas.get());

  // All transactions need to get committed or aborted.
  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 3) {
    std::cerr << "Usage: example_rendering <scene_file> <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* scene_file = argv[1];
  const char* mdl_path   = argv[2];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get(), scene_file);
```

```
  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.19 example_rtmp_server.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <fstream>
#include <iostream>
#include <vector>


class Buffer : public mi::base::Interface_implement<mi::neuraylib::IBuffer>
{
public:
  const mi::Uint8* get_data() const { return &m_buffer[0]; }

  mi::Size get_data_size() const { return m_buffer.size(); }

  Buffer( const std::vector<mi::Uint8>& content) { m_buffer = content; }

private:
  std::vector<mi::Uint8> m_buffer;
};

class Response_handler : public mi::base::Interface_implement<mi::http::IResponse_handler>
{
public:
  void handle( mi::http::IConnection* connection)
  {
    mi::http::IResponse* iresponse( connection->get_response());
    iresponse->set_header( "Content-Type", "application/x-shockwave-flash");
  }
};

class Request_handler : public mi::base::Interface_implement<mi::http::IRequest_handler>
{
public:
  Request_handler( const char* swf_file) : m_swf_file( swf_file) { }

  bool handle( mi::http::IConnection* connection)
  {
    std::ifstream file( m_swf_file, std::ios::in|std::ios::binary|std::ios::ate);
    check_success( file);

    std::ifstream::pos_type size = file.tellg();
    std::vector<mi::Uint8> data( static_cast<mi::Size>( size));
    file.seekg( 0, std::ios::beg);
    file.read( reinterpret_cast<char*>( &data[0]), size);
    file.close();

    mi::base::Handle<mi::neuraylib::IBuffer> buffer( new Buffer( data));
```

```
    connection->enqueue( buffer.get());
    return true;
  }

private:
  const char* m_swf_file;
};


class Play_event_handler : public mi::base::Interface_implement<mi::rtmp::IPlay_event_handler>
{
public:
 bool handle( bool is_start, mi::rtmp::IStream* stream, mi::neuraylib::IVideo_data** out)
  {
    if( is_start) {
      check_success( stream->use_codec( "screen video"));
    mi::base::Handle<mi::neuraylib::IVideo_encoder> codec( stream->get_video_codec());
      check_success( codec->init( 512, 384, out) == 0);
    }
    else {
    mi::base::Handle<mi::neuraylib::IVideo_encoder> codec( stream->get_video_codec());
      check_success( codec->close( out) == 0);
    }
    return true;
  }
};

class Frame_event_handler : public mi::base::Interface_implement<mi::rtmp::IFrame_event_handler>
{
public:
  bool handle(
   mi::rtmp::IStream* stream, mi::neuraylib::IVideo_data** out, bool send_queue_is_full)
  {
    if (send_queue_is_full) // we do not want to increase buffering
      return true;
   mi::base::Handle<mi::neuraylib::IVideo_encoder> codec( stream->get_video_codec());
    mi::neuraylib::ICanvas* canvas = 0;
    {
      mi::base::Lock::Block block( &m_cached_canvas_lock);
      canvas = m_cached_canvas.get();
      if ( !canvas)
        return true;
      canvas->retain();
    }
    bool result = (codec->encode_canvas( canvas, out) == 0);
    canvas->release();
    return result;
  }
  void update_canvas( mi::neuraylib::ICanvas* new_canvas)
  {
    mi::base::Lock::Block block( &m_cached_canvas_lock);
    m_cached_canvas = make_handle_dup( new_canvas);
  }
private:
  mi::base::Lock m_cached_canvas_lock;
  mi::base::Handle<mi::neuraylib::ICanvas> m_cached_canvas;
};
```

```
class Render_event_handler : public mi::base::Interface_implement<mi::rtmp::IRender_event_handler>
{
public:
  Render_event_handler(
   mi::neuraylib::INeuray* neuray, mi::neuraylib::IScope* scope, Frame_event_handler* handler)
    : m_neuray( neuray, mi::base::DUP_INTERFACE),
     m_scope( scope, mi::base::DUP_INTERFACE),
     m_frame_handler( handler, mi::base::DUP_INTERFACE)
  {
   mi::base::Handle<mi::neuraylib::ITransaction> transaction( m_scope->create_transaction());
    {
      mi::base::Handle<mi::neuraylib::IScene> scene(
        transaction->edit<mi::neuraylib::IScene>( "the_scene"));
      m_render_context = scene->create_render_context( transaction.get(), "iray");
      check_success( m_render_context.is_valid_interface());
    mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
      scheduler_mode->set_c_str( "interactive");
      m_render_context->set_option( "scheduler_mode", scheduler_mode.get());
      mi::base::Handle<mi::IFloat32> interval( transaction->create<mi::IFloat32>());
      interval->set_value( 0.1f);
      m_render_context->set_option( "interactive_update_interval", interval.get());
    }
    transaction->commit();
  }

  bool handle( mi::rtmp::IStream* /*stream*/)
  {
   mi::base::Handle<mi::neuraylib::ITransaction> transaction( m_scope->create_transaction());
    {
      mi::base::Handle<mi::neuraylib::IImage_api> image_api(
        m_neuray->get_api_component<mi::neuraylib::IImage_api>());
      mi::base::Handle<mi::neuraylib::IRender_target> render_target(
        new Render_target( image_api.get(), "Color", 512, 384));
      check_success(
        m_render_context->render( transaction.get(), render_target.get(), 0) >= 0);

     mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
      m_frame_handler->update_canvas( canvas.get());
    }
    transaction->commit();
    return true;
  }

private:
  mi::base::Handle<mi::neuraylib::INeuray> m_neuray;
  mi::base::Handle<mi::neuraylib::IScope> m_scope;
  mi::base::Handle<Frame_event_handler> m_frame_handler;
  mi::base::Handle<mi::neuraylib::IRender_context> m_render_context;
};

class Stream_event_handler : public mi::base::Interface_implement<mi::rtmp::IStream_event_handler>
{
public:
  Stream_event_handler( mi::neuraylib::INeuray* neuray, mi::neuraylib::IScope* scope)
   : m_neuray( neuray, mi::base::DUP_INTERFACE), m_scope( scope, mi::base::DUP_INTERFACE) { }
```

```
  bool handle(
    bool is_create, mi::rtmp::IStream* stream,
    const mi::IData* /*command_arguments*/)
  {
    if( is_create) {
      mi::base::Handle<mi::rtmp::IPlay_event_handler> play_event_handler(
        new Play_event_handler());
      stream->register_play_event_handler( play_event_handler.get());
    mi::base::Handle<Frame_event_handler> frame_event_handler( new Frame_event_handler());
      mi::base::Handle<mi::rtmp::IRender_event_handler> render_event_handler(
     new Render_event_handler( m_neuray.get(), m_scope.get(),frame_event_handler.get()));
      stream->register_render_event_handler( render_event_handler.get());
      stream->register_frame_event_handler( frame_event_handler.get());
    }
    return true;
  }

private:
  mi::base::Handle<mi::neuraylib::INeuray> m_neuray;
  mi::base::Handle<mi::neuraylib::IScope> m_scope;
};

class Call_event_handler : public mi::base::Interface_implement<mi::rtmp::ICall_event_handler>
{
public:
 Call_event_handler( mi::neuraylib::IScope* scope) : m_scope( scope, mi::base::DUP_INTERFACE) { }

  bool handle(
    mi::rtmp::IConnection* /*connection*/,
    const char* /*procedure_name*/,
    const mi::IData* /*command_arguments*/,
    const mi::IData* user_arguments,
    mi::IData** /*response_arguments*/)
  {
   mi::base::Handle<mi::neuraylib::ITransaction> transaction( m_scope->create_transaction());
    {
      // The "camera" name matches the camera in main.mi in the examples directory.
      mi::base::Handle<mi::neuraylib::ICamera> camera(
        transaction->edit<mi::neuraylib::ICamera>( "camera"));
      check_success( camera.is_valid_interface());
    mi::base::Handle<const mi::IMap> imap( user_arguments->get_interface<const mi::IMap>());
      check_success( imap.is_valid_interface());
    mi::base::Handle<const mi::ISint32> pan_x( imap->get_value<mi::ISint32>( "pan_x"));
      if ( pan_x) {
        mi::Float64 x = camera->get_offset_x();
        camera->set_offset_x( x - pan_x->get_value<mi::Sint32>());
        // The example client also demonstrates how to send/parse a double.
        mi::base::Handle<const mi::IFloat64> pan_xd(
          imap->get_value<mi::IFloat64>( "pan_xd"));
        if( pan_xd) {
          mi::Float64 xd = pan_xd->get_value<mi::Float64>();
          check_success( mi::Sint32(xd) == pan_x->get_value<mi::Sint32>());
        }
      }
    mi::base::Handle<const mi::ISint32> pan_y( imap->get_value<mi::ISint32>( "pan_y"));
      if( pan_y) {
```

```
        mi::Float64 y = camera->get_offset_y();
        camera->set_offset_y( y - pan_y->get_value<mi::Sint32>());
      }
      // Demonstrate getting a bool from the example client
      mi::base::Handle<const mi::IBoolean> dir(
        imap->get_value<mi::IBoolean>( "going_right"));
      if ( dir) {
        bool going_right = dir->get_value<bool>();
        going_right = !going_right; // avoid compiler warning
      }
    }
    transaction->commit();
    return true;
  }

private:
  mi::base::Handle<mi::neuraylib::IScope> m_scope;
};

class Connect_event_handler : public mi::base::Interface_implement<mi::rtmp::IConnect_event_handler
{
public:
 Connect_event_handler( mi::neuraylib::INeuray* neuray, mi::neuraylib::IScope* scope)
  : m_neuray( neuray, mi::base::DUP_INTERFACE), m_scope( scope, mi::base::DUP_INTERFACE) { }

  bool handle(
    bool is_create, mi::rtmp::IConnection* connection,
    const mi::IData* /*command_arguments*/,
    const mi::IData* /*user_arguments*/)
  {
    if( is_create) {
      mi::base::Handle<mi::rtmp::IStream_event_handler> stream_event_handler(
        new Stream_event_handler( m_neuray.get(), m_scope.get()));
      connection->register_stream_event_handler( stream_event_handler.get());
      mi::base::Handle<mi::rtmp::ICall_event_handler> call_event_handler(
        new Call_event_handler( m_scope.get()));
    connection->register_remote_call_handler( call_event_handler.get(), "moveCamera");
    }
    return true;
  }

private:
  mi::base::Handle<mi::neuraylib::INeuray> m_neuray;
  mi::base::Handle<mi::neuraylib::IScope> m_scope;
};

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
 // Also load the default video codec plugin which will be used to encode the rendered frames.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
```

```
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
  check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "screen_video" MI_BASE_DLL_FILE_EXT) == 0);
}

void prepare_rendering(
  mi::neuraylib::INeuray* neuray, const char* scene_file)
{
  // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::IString> uri( import_api->convert_filename_to_uri( scene_file));
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), uri->get_c_str()));
  check_success( import_result->get_error_number() == 0);

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());

  // And store it in the database such that the render loop can later access it
  transaction->store( scene.get(), "the_scene");
  transaction->commit();
}

void run_http_and_rtmp_server(
  mi::neuraylib::INeuray* neuray, const char* port, const char* swf_file)
{
  // Create an HTTP server instance
  mi::base::Handle<mi::http::IFactory> http_factory(
    neuray->get_api_component<mi::http::IFactory>());
  mi::base::Handle<mi::http::IServer> http_server(
    http_factory->create_server());

  // Install our HTTP request and response handlers
  mi::base::Handle<mi::http::IRequest_handler> request_handler(
    new Request_handler( swf_file));
  http_server->install( request_handler.get());
  mi::base::Handle<mi::http::IResponse_handler> response_handler(
    new Response_handler());
```

```
  http_server->install( response_handler.get());

  // Assemble HTTP server address
  const char* ip = "0.0.0.0:";
  char address[255];
  address[0] = '\0';
  strncat( address, ip, sizeof(address) - 1);
  strncat( address, port, sizeof(address) - 1 - strlen(address));

  // Start HTTP server
  http_server->start( address);

  // Create an RTMP server instance
  mi::base::Handle<mi::rtmp::IFactory> rtmp_factory(
    neuray->get_api_component<mi::rtmp::IFactory>());
  mi::base::Handle<mi::rtmp::IServer> rtmp_server( rtmp_factory->create_server());

  // Install our HTTP connect handler
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::rtmp::IConnect_event_handler> connect_handler(
    new Connect_event_handler( neuray, scope.get()));
  rtmp_server->install( connect_handler.get());

  // Start RTMP server
  rtmp_server->start( "0.0.0.0:1935");

  // Run both servers for fixed time interval
  sleep_seconds( 30);
  http_server->shutdown();
  rtmp_server->shutdown();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 5) {
   std::cerr << "Usage: example_rtmp_server <swf_file> <scene_file> <mdl_path> <port>"
         << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* swf_file   = argv[1];
  const char* scene_file = argv[2];
  const char* mdl_path   = argv[3];
  const char* port       = argv[4];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
```

```
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Set up the scene
  prepare_rendering( neuray.get(), scene_file);

  // Serve video stream via RTMP server
  run_http_and_rtmp_server( neuray.get(), port, swf_file);

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.20 example_rtmp_server.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/*****************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="horizontal"
  initialize="init()" xmlns:local="*">

<mx:Script>
  <![CDATA[
    import mx.core.Application;

    public function init():void {
      vidplayer.makeConnection("rtmp://" + getHost());
    }

    public function getHost():String  {
      var location:String = Application.application.url;
      var components:Array = location.split("/");
      if (components.length < 3)
        return "localhost";
      var host_port:Array = components[2].split(":");
      if (host_port.length <= 1)
        return "localhost";
      return host_port[0];
    }
  ]]>
</mx:Script>

<!-- refer to the actionscript object -->
<local:example_rtmp_server_actionscript includeInLayout="true" id="vidplayer" width="1024" height="
</mx:Application>
```

20.21 example_rtmp_server_actionscript.as

```
/*****************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
* Germany. All rights reserved
*****************************************************************************/

package {
  import flash.events.MouseEvent;
  import flash.events.NetStatusEvent;
  import flash.events.SecurityErrorEvent;
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import mx.core.Application;
  import mx.core.UIComponent;

  public class example_rtmp_server_actionscript extends UIComponent {
    private var streamName:String = "example_rtmp_server";
    public var connection:NetConnection = null;
    private var video:Video = null;
    private var mystream:NetStream = null;
    private var client:Object = null;
    private var mouseButton:Boolean = false;
    private var mousePosX:int = 0;
    private var mousePosY:int = 0;

    public function example_rtmp_server_actionscript() {
      super();
      this.addEventListener(MouseEvent.MOUSE_DOWN, this.onMouseDown);
    }

    public function makeConnection(url:String):void {
      if (connection != null) {
        mystream = null;
        connection.close();
      } else {
        connection = new NetConnection();
      }
      connection.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    connection.addEventListener(SecurityErrorEvent.SECURITY_ERROR, securityErrorHandler);
      var args:Object = new Object();
      args["resolution_x"] = floor16(this.width).toString();
      args["resolution_y"] = floor16(this.height).toString();
      connection.connect(url,args);
    }

    private function floor16(val:int):int  {
      return int(val/16) * 16;
    }

    public function closeConnection():void {
      if (connection != null) {
        mystream = null;
        connection.close();
      }
    }
```

```
private function netStatusHandler(event:NetStatusEvent):void {
  switch (event.info.code) {
    case "NetConnection.Connect.Success":
      connectStream();
      break;
    case "NetStream.Play.StreamNotFound":
      trace("Stream not found: " + streamName);
      break;
  }
}

private function securityErrorHandler(event:SecurityErrorEvent):void {
  trace("securityErrorHandler: " + event);
}

private function connectStream():void {
  mystream = new NetStream(connection);
  mystream.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
  if (video == null) {
    video = new Video(this.width,this.height);
    video.smoothing = true;
  }
  video.attachNetStream(mystream);
  addChild(video);
  mystream.play(streamName);
}

public function onMouseDown(event: MouseEvent):void {
  var x: int = event.stageX - (event.target as UIComponent).parent.x;
  var y: int = event.stageY - (event.target as UIComponent).parent.y;
  mousePosX = x;
  mousePosY = y;
  Application.application.addEventListener(MouseEvent.MOUSE_UP, this.onMouseUp);
Application.application.addEventListener(MouseEvent.MOUSE_MOVE, this.onMouseMove);
  mouseButton = true;
}

public function onMouseUp(event: MouseEvent):void {
  if (mouseButton) {
    mouseButton = false;
 Application.application.removeEventListener(MouseEvent.MOUSE_UP, this.onMouseUp);
 Application.application.removeEventListener(MouseEvent.MOUSE_MOVE, this.onMouseMove);
  }
}

public function onMouseMove(event: MouseEvent):void
{
  var x: int = event.stageX - (event.target as UIComponent).parent.x;
  var y: int = event.stageY - (event.target as UIComponent).parent.y;
  if (mouseButton && connection && connection.connected && mystream) {
    var diff_x:int = x-mousePosX;
    var diff_y:int = y-mousePosY;
    var args:Object = new Object();
    if (diff_x != 0) args["pan_x"] = diff_x;
    if (diff_y != 0) args["pan_y"] = -diff_y;
    if (diff_x || diff_y) {
```

```
          // For demonstration purposes also send a double..
          args["pan_xd"] = (diff_x < 0) ? diff_x - 0.1 : diff_x + 0.1
          // ..and some bool
          args["going_right"] = diff_x > 0 ? true : false;
          connection.call("moveCamera",null,args);
        }
        mousePosX = x;
        mousePosY = y;
      }
    }
  }
}
```

20.22 example_scene.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <iostream>

#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

const mi::Uint32 cube_n_points    = 8;
const mi::Uint32 cube_n_normals   = 6;
const mi::Uint32 cube_n_uvs       = 4;
const mi::Uint32 cube_n_triangles = 12;

mi::Float32_3 cube_points[cube_n_points] = {
  mi::Float32_3( -0.5, -0.5, -0.5),
  mi::Float32_3( -0.5, -0.5,  0.5),
  mi::Float32_3( -0.5,  0.5, -0.5),
  mi::Float32_3( -0.5,  0.5,  0.5),
  mi::Float32_3(  0.5, -0.5, -0.5),
  mi::Float32_3(  0.5, -0.5,  0.5),
  mi::Float32_3(  0.5,  0.5, -0.5),
  mi::Float32_3(  0.5,  0.5,  0.5) };

mi::Float32_3 cube_normals[cube_n_normals] = {
  mi::Float32_3(  0.0,  0.0, -1.0),
  mi::Float32_3(  0.0, -1.0,  0.0),
  mi::Float32_3( -1.0,  0.0,  0.0),
  mi::Float32_3(  0.0,  0.0,  1.0),
  mi::Float32_3(  0.0,  1.0,  0.0),
  mi::Float32_3(  1.0,  0.0,  0.0) };

mi::Float32_2 cube_uvs[cube_n_uvs] = {
  mi::Float32_2(  0.0,  0.0),
  mi::Float32_2(  1.0,  0.0),
  mi::Float32_2(  0.0,  1.0),
  mi::Float32_2(  1.0,  1.0) };

mi::neuraylib::Triangle_point_indices cube_mesh_connectivity[cube_n_triangles] = {
  mi::neuraylib::Triangle_point_indices(  0,  2,  6),
  mi::neuraylib::Triangle_point_indices(  6,  4,  0),
  mi::neuraylib::Triangle_point_indices(  0,  4,  5),
  mi::neuraylib::Triangle_point_indices(  5,  1,  0),
  mi::neuraylib::Triangle_point_indices(  0,  1,  3),
  mi::neuraylib::Triangle_point_indices(  3,  2,  0),
  mi::neuraylib::Triangle_point_indices(  1,  5,  7),
  mi::neuraylib::Triangle_point_indices(  7,  3,  1),
  mi::neuraylib::Triangle_point_indices(  2,  3,  7),
  mi::neuraylib::Triangle_point_indices(  7,  6,  2),
  mi::neuraylib::Triangle_point_indices(  4,  6,  7),
  mi::neuraylib::Triangle_point_indices(  7,  5,  4) };
```

```
mi::neuraylib::Triangle_point_indices cube_normal_connectivity[cube_n_triangles] = {
  mi::neuraylib::Triangle_point_indices(  0,  0,  0),
  mi::neuraylib::Triangle_point_indices(  0,  0,  0),
  mi::neuraylib::Triangle_point_indices(  1,  1,  1),
  mi::neuraylib::Triangle_point_indices(  1,  1,  1),
  mi::neuraylib::Triangle_point_indices(  2,  2,  2),
  mi::neuraylib::Triangle_point_indices(  2,  2,  2),
  mi::neuraylib::Triangle_point_indices(  3,  3,  3),
  mi::neuraylib::Triangle_point_indices(  3,  3,  3),
  mi::neuraylib::Triangle_point_indices(  4,  4,  4),
  mi::neuraylib::Triangle_point_indices(  4,  4,  4),
  mi::neuraylib::Triangle_point_indices(  5,  5,  5),
  mi::neuraylib::Triangle_point_indices(  5,  5,  5) };

mi::neuraylib::Triangle_point_indices cube_uv_connectivity[cube_n_triangles] = {
  mi::neuraylib::Triangle_point_indices(  0,  1,  3),
  mi::neuraylib::Triangle_point_indices(  3,  2,  0),
  mi::neuraylib::Triangle_point_indices(  0,  1,  3),
  mi::neuraylib::Triangle_point_indices(  3,  2,  0),
  mi::neuraylib::Triangle_point_indices(  0,  1,  3),
  mi::neuraylib::Triangle_point_indices(  3,  2,  0),
  mi::neuraylib::Triangle_point_indices(  0,  1,  3),
  mi::neuraylib::Triangle_point_indices(  3,  2,  0),
  mi::neuraylib::Triangle_point_indices(  0,  1,  3),
  mi::neuraylib::Triangle_point_indices(  3,  2,  0),
  mi::neuraylib::Triangle_point_indices(  0,  1,  3),
  mi::neuraylib::Triangle_point_indices(  3,  2,  0) };

const mi::Uint32 ground_n_points    = 4;
const mi::Uint32 ground_n_normals   = 1;
const mi::Uint32 ground_n_uvs       = 4;
const mi::Uint32 ground_n_triangles = 2;

mi::Float32_3 ground_points[ground_n_points] = {
  mi::Float32_3( -2.0,  0.0, -2.0),
  mi::Float32_3( -2.0,  0.0,  2.0),
  mi::Float32_3(  2.0,  0.0,  2.0),
  mi::Float32_3(  2.0,  0.0, -2.0) };

mi::Float32_3 ground_normals[ground_n_normals] = {
  mi::Float32_3(  0.0,  1.0,  0.0) };

mi::Float32_2 ground_uvs[ground_n_uvs] = {
  mi::Float32_2(  0.0,  1.0),
  mi::Float32_2(  0.0,  0.0),
  mi::Float32_2(  1.0,  0.0),
  mi::Float32_2(  1.0,  1.0) };

mi::neuraylib::Triangle_point_indices ground_mesh_connectivity[ground_n_triangles] = {
  mi::neuraylib::Triangle_point_indices(  0,  1,  2),
  mi::neuraylib::Triangle_point_indices(  2,  3,  0) };

mi::neuraylib::Triangle_point_indices ground_normal_connectivity[ground_n_triangles] = {
  mi::neuraylib::Triangle_point_indices(  0,  0,  0),
  mi::neuraylib::Triangle_point_indices(  0,  0,  0) };

mi::neuraylib::Triangle_point_indices ground_uv_connectivity[ground_n_triangles] = {
```

```
  mi::neuraylib::Triangle_point_indices(  0,  1,  2),
  mi::neuraylib::Triangle_point_indices(  2,  3,  0) };

void create_flag( mi::neuraylib::IAttribute_set* attribute_set, const char* name, bool value)
{
  mi::base::Handle<mi::IBoolean> attribute(
    attribute_set->create_attribute<mi::IBoolean>( name, "Boolean"));
  attribute->set_value( value);
}

mi::neuraylib::IScene* create_scene(
  mi::neuraylib::INeuray* neuray,
  mi::neuraylib::ITransaction* transaction)
{
  mi::base::Handle<mi::neuraylib::IMdl_factory> mdl_factory(
    neuray->get_api_component<mi::neuraylib::IMdl_factory>());
  mi::base::Handle<mi::neuraylib::IValue_factory> value_factory(
    mdl_factory->create_value_factory( transaction));
  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction));

  {
    // Import the MDL module "main"
    mi::base::Handle<mi::neuraylib::IImport_api> import_api(
      neuray->get_api_component<mi::neuraylib::IImport_api>());
    check_success( import_api.is_valid_interface());
    mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
      import_api->import_elements( transaction, "file:${shader}/main.mdl"));
    check_success( import_result->get_error_number() == 0);
  }
  {
    // Create the options "options"
    mi::base::Handle<mi::neuraylib::IOptions> options(
      transaction->create<mi::neuraylib::IOptions>( "Options"));
    transaction->store( options.get(), "options");
  }
  {
    // Create the MDL material instance "yellow_material" used by "cube_instance"
    mi::base::Handle<mi::neuraylib::IValue> value(
      value_factory->create_color( 0.942f, 0.807216f, 0.33441f));
    mi::base::Handle<mi::neuraylib::IExpression> expression(
      expression_factory->create_constant( value.get()));
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "tint", expression.get());
    mi::base::Handle<const mi::neuraylib::IFunction_definition> definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::main::diffuse_material(color)"));
    mi::base::Handle<mi::neuraylib::IFunction_call> material(
      definition->create_function_call( arguments.get()));
    check_success( material.get());
    transaction->store( material.get(), "yellow_material");
  }
  {
    // Create the MDL material instance "grey_material" used by "ground_instance"
    mi::base::Handle<mi::neuraylib::IValue> value(
      value_factory->create_color( 0.306959f, 0.306959f, 0.306959f));
```

```
    mi::base::Handle<mi::neuraylib::IExpression> expression(
      expression_factory->create_constant( value.get()));
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "tint", expression.get());
    mi::base::Handle<const mi::neuraylib::IFunction_definition> definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::main::diffuse_material(color)"));
    mi::base::Handle<mi::neuraylib::IFunction_call> material(
      definition->create_function_call( arguments.get()));
    check_success( material.get());
    transaction->store( material.get(), "grey_material");
  }
  {
    // Create the MDL material instance "white_light" used by "light"
    mi::base::Handle<mi::neuraylib::IValue> value(
      value_factory->create_color( 1000.0f, 1000.0f, 1000.0f));
    mi::base::Handle<mi::neuraylib::IExpression> expression(
      expression_factory->create_constant( value.get()));
    mi::base::Handle<mi::neuraylib::IExpression_list> arguments(
      expression_factory->create_expression_list());
    arguments->add_expression( "tint", expression.get());
    mi::base::Handle<const mi::neuraylib::IFunction_definition> definition(
      transaction->access<mi::neuraylib::IFunction_definition>(
        "mdl::main::diffuse_light(color)"));
    mi::base::Handle<mi::neuraylib::IFunction_call> material(
      definition->create_function_call( arguments.get()));
    check_success( material.get());
    transaction->store( material.get(), "white_light");
  }
  {
    // Create the triangle mesh "cube"
    mi::base::Handle<mi::neuraylib::ITriangle_mesh> mesh(
      transaction->create<mi::neuraylib::ITriangle_mesh>( "Triangle_mesh"));
    create_flag( mesh.get(), "visible", true);
    create_flag( mesh.get(), "reflection_cast", true);
    create_flag( mesh.get(), "reflection_recv", true);
    create_flag( mesh.get(), "refraction_cast", true);
    create_flag( mesh.get(), "refraction_recv", true);
    create_flag( mesh.get(), "shadow_cast", true);
    create_flag( mesh.get(), "shadow_recv", true);

    // Set point data and mesh connectivity
    mesh->reserve_points( cube_n_points);
    for( mi::Uint32 i = 0; i < cube_n_points; ++i)
      mesh->append_point( cube_points[i]);
    mesh->reserve_triangles( cube_n_triangles);
    for( mi::Uint32 i = 0; i < cube_n_triangles; ++i)
      mesh->append_triangle( cube_mesh_connectivity[i]);

    // Set normal data and normal connectivity
    mi::base::Handle<mi::neuraylib::ITriangle_connectivity> normal_connectivity(
      mesh->create_connectivity());
    for( mi::Uint32 i = 0; i < cube_n_triangles; ++i)
      normal_connectivity->set_triangle_indices(
        mi::neuraylib::Triangle_handle( i), cube_normal_connectivity[i]);
    mi::base::Handle<mi::neuraylib::IAttribute_vector> normals(
```

```
    normal_connectivity->create_attribute_vector( mi::neuraylib::ATTR_NORMAL));
  for( mi::Uint32 i = 0; i < cube_n_normals; ++i)
    normals->append_vector3( cube_normals[i]);
 check_success( normal_connectivity->attach_attribute_vector( normals.get()) == 0);
  check_success( mesh->attach_connectivity( normal_connectivity.get()) == 0);

  // Set uv data and uv connectivity
  mi::base::Handle<mi::neuraylib::ITriangle_connectivity> uv_connectivity(
    mesh->create_connectivity());
  for( mi::Uint32 i = 0; i < cube_n_triangles; ++i)
    uv_connectivity->set_triangle_indices(
      mi::neuraylib::Triangle_handle( i), cube_uv_connectivity[i]);
  mi::base::Handle<mi::neuraylib::IAttribute_vector> uvs(
    uv_connectivity->create_attribute_vector( mi::neuraylib::ATTR_TEXTURE, 2));
  for( mi::Uint32 i = 0; i < cube_n_uvs; ++i)
    uvs->append_float32( &cube_uvs[i][0], 2);
  check_success( uv_connectivity->attach_attribute_vector( uvs.get()) == 0);
  check_success( mesh->attach_connectivity( uv_connectivity.get()) == 0);

  transaction->store( mesh.get(), "cube");
}
{
  // Create the instance "cube_instance" referencing "cube"
  mi::base::Handle<mi::neuraylib::IInstance> instance(
    transaction->create<mi::neuraylib::IInstance>( "Instance"));
  check_success( instance->attach( "cube") == 0);
  mi::Float64_4_4 matrix( 1.0f);
  matrix.translate( 1.3f, -0.5f, 1.0f);
  instance->set_matrix( matrix);
  mi::base::Handle<mi::IRef> material(
    instance->create_attribute<mi::IRef>( "material", "Ref"));
  check_success( material->set_reference( "yellow_material") == 0);
  transaction->store( instance.get(), "cube_instance");
}
{
  // Create the triangle mesh "ground"
  mi::base::Handle<mi::neuraylib::ITriangle_mesh> mesh(
    transaction->create<mi::neuraylib::ITriangle_mesh>( "Triangle_mesh"));
  create_flag( mesh.get(), "visible", true);
  create_flag( mesh.get(), "reflection_cast", true);
  create_flag( mesh.get(), "reflection_recv", true);
  create_flag( mesh.get(), "refraction_cast", true);
  create_flag( mesh.get(), "refraction_recv", true);
  create_flag( mesh.get(), "shadow_cast", true);
  create_flag( mesh.get(), "shadow_recv", true);

  // Set point data and mesh connectivity
  mesh->reserve_points( ground_n_points);
  for( mi::Uint32 i = 0; i < ground_n_points; ++i)
    mesh->append_point( ground_points[i]);
  mesh->reserve_triangles( ground_n_triangles);
  for( mi::Uint32 i = 0; i < ground_n_triangles; ++i)
    mesh->append_triangle( ground_mesh_connectivity[i]);

  // Set normal data and normal connectivity
  mi::base::Handle<mi::neuraylib::ITriangle_connectivity> normal_connectivity(
    mesh->create_connectivity());
```

```
    for( mi::Uint32 i = 0; i < ground_n_triangles; ++i)
      normal_connectivity->set_triangle_indices(
        mi::neuraylib::Triangle_handle( i), ground_normal_connectivity[i]);
    mi::base::Handle<mi::neuraylib::IAttribute_vector> normals(
      normal_connectivity->create_attribute_vector( mi::neuraylib::ATTR_NORMAL));
    for( mi::Uint32 i = 0; i < ground_n_normals; ++i)
      normals->append_vector3( ground_normals[i]);
   check_success( normal_connectivity->attach_attribute_vector( normals.get()) == 0);
    check_success( mesh->attach_connectivity( normal_connectivity.get()) == 0);

    // Set uv data and uv connectivity
    mi::base::Handle<mi::neuraylib::ITriangle_connectivity> uv_connectivity(
      mesh->create_connectivity());
    for( mi::Uint32 i = 0; i < ground_n_triangles; ++i)
      uv_connectivity->set_triangle_indices(
        mi::neuraylib::Triangle_handle( i), ground_uv_connectivity[i]);
    mi::base::Handle<mi::neuraylib::IAttribute_vector> uvs(
      uv_connectivity->create_attribute_vector( mi::neuraylib::ATTR_TEXTURE, 2));
    for( mi::Uint32 i = 0; i < ground_n_uvs; ++i)
      uvs->append_float32( &ground_uvs[i][0], 2);
    check_success( uv_connectivity->attach_attribute_vector( uvs.get()) == 0);
    check_success( mesh->attach_connectivity( uv_connectivity.get()) == 0);

    transaction->store( mesh.get(), "ground");
  }
  {
    // Create the instance "ground_instance" referencing "ground"
    mi::base::Handle<mi::neuraylib::IInstance> instance(
      transaction->create<mi::neuraylib::IInstance>( "Instance"));
    check_success( instance->attach( "ground") == 0);
    mi::base::Handle<mi::IRef> material(
      instance->create_attribute<mi::IRef>( "material", "Ref"));
    check_success( material->set_reference( "grey_material") == 0);
    transaction->store( instance.get(), "ground_instance");
  }
  {
    // Create the light "light"
    mi::base::Handle<mi::neuraylib::ILight> light(
      transaction->create<mi::neuraylib::ILight>( "Light"));
    create_flag( light.get(), "shadow_cast", true);
    mi::base::Handle<mi::IRef> material(
      light->create_attribute<mi::IRef>( "material", "Ref"));
    check_success( material->set_reference( "white_light") == 0);
    transaction->store( light.get(), "light");
  }
  {
    // Create the instance "light_instance" referencing "light"
    mi::base::Handle<mi::neuraylib::IInstance> instance(
      transaction->create<mi::neuraylib::IInstance>( "Instance"));
    check_success( instance->attach( "light") == 0);
    mi::Float64_4_4 matrix( 1.0f);
    matrix.translate( 5.1f, -7.3f, -1.6f);
    instance->set_matrix( matrix);
    transaction->store( instance.get(), "light_instance");
  }
  {
    // Create the camera "camera"
```

```
    mi::base::Handle<mi::neuraylib::ICamera> camera(
      transaction->create<mi::neuraylib::ICamera>( "Camera"));
    camera->set_focal( 50.0f);
    camera->set_aperture( 44.0f);
    camera->set_aspect( 1.33333f);
    camera->set_resolution_x( 512);
    camera->set_resolution_y( 384);
    camera->set_clip_min( 0.1);
    camera->set_clip_max( 1000);
    transaction->store( camera.get(), "camera");
  }
  {
    // Create the instance "camera_instance" referencing "camera"
    mi::base::Handle<mi::neuraylib::IInstance> instance(
      transaction->create<mi::neuraylib::IInstance>( "Instance"));
    check_success( instance->attach( "camera") == 0);
    mi::Float64_4_4 matrix(
      0.68826f,  0.37107f, -0.623382f, 0.0f,
      0.00000f,  0.85929f,  0.511493f, 0.0f,
      0.72546f, -0.35204f,  0.591414f, 0.0f,
      0.00000f,  0.00000f, -6.256200f, 1.0f);
    instance->set_matrix( matrix);
    transaction->store( instance.get(), "camera_instance");
  }
  {
    // Create the group "rootgroup" containing all instances
    mi::base::Handle<mi::neuraylib::IGroup> group(
      transaction->create<mi::neuraylib::IGroup>( "Group"));
    check_success( group->attach( "cube_instance"  ) == 0);
    check_success( group->attach( "ground_instance") == 0);
    check_success( group->attach( "light_instance" ) == 0);
    check_success( group->attach( "camera_instance") == 0);
    transaction->store( group.get(), "rootgroup");
  }

  // Create the scene object itself and return it.
 mi::neuraylib::IScene* scene = transaction->create<mi::neuraylib::IScene>( "Scene");
  scene->set_rootgroup( "rootgroup");
  scene->set_camera_instance( "camera_instance");
  scene->set_options( "options");
  return scene;
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc.is_valid_interface());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO and Iray Photoreal plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
```

```
}

void rendering( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Create the scene
 mi::base::Handle<mi::neuraylib::IScene> scene( create_scene( neuray, transaction.get()));
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_scene.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_scene <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
```

```
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.23 example_shared.h

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#ifndef EXAMPLE_SHARED_H
#define EXAMPLE_SHARED_H

#include <cstdio>
#include <cstdlib>

#include <mi/neuraylib.h>

#ifdef MI_PLATFORM_WINDOWS
#include <mi/base/miwindows.h>
#else
#include <dlfcn.h>
#include <unistd.h>
#endif

#include "authentication.h"

void* g_dso_handle = 0;

void keep_console_open() {
#ifdef MI_PLATFORM_WINDOWS
  if( IsDebuggerPresent()) {
    fprintf( stderr, "Press enter to continue . . . \n");
    fgetc( stdin);
  }
#endif // MI_PLATFORM_WINDOWS
}

#define check_success( expr) \
  do { \
    if( !(expr)) { \
    fprintf( stderr, "Error in file %s, line %u: \"%s\".\n", __FILE__, __LINE__, #expr); \
      keep_console_open(); \
      exit( EXIT_FAILURE); \
    } \
  } while( false)

void check_start_success( mi::Sint32 result)
{
  if( result == 0)
    return;
 fprintf( stderr, "mi::neuraylib::INeuray::start() failed with return code %d.\n", result);
 fprintf( stderr, "Typical failure reasons are related to authentication, see the\n");
  fprintf( stderr, "documentation of this method for details.\n");
  keep_console_open();
  exit( EXIT_FAILURE);
}

#ifdef MI_PLATFORM_WINDOWS
#ifdef UNICODE
```

```
#define FMT_LPTSTR "%ls"
#else // UNICODE
#define FMT_LPTSTR "%s"
#endif // UNICODE
#endif // MI_PLATFORM_WINDOWS

mi::neuraylib::INeuray* load_and_get_ineuray( const char* filename = 0)
{
  if( !filename)
    filename = "libneuray" MI_BASE_DLL_FILE_EXT;
#ifdef MI_PLATFORM_WINDOWS
  void* handle = LoadLibraryA((LPSTR) filename);
  if( !handle) {
    LPTSTR buffer = 0;
    LPTSTR message = TEXT("unknown failure");
    DWORD error_code = GetLastError();
    if( FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
      FORMAT_MESSAGE_IGNORE_INSERTS, 0, error_code,
      MAKELANGID( LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR) &buffer, 0, 0))
      message = buffer;
   fprintf( stderr, "Failed to load library (%u): " FMT_LPTSTR, error_code, message);
    if( buffer)
      LocalFree( buffer);
    return 0;
  }
  void* symbol = GetProcAddress((HMODULE) handle, "mi_factory");
  if( !symbol) {
    LPTSTR buffer = 0;
    LPTSTR message = TEXT("unknown failure");
    DWORD error_code = GetLastError();
    if( FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
      FORMAT_MESSAGE_IGNORE_INSERTS, 0, error_code,
      MAKELANGID( LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR) &buffer, 0, 0))
      message = buffer;
    fprintf( stderr, "GetProcAddress error (%u): " FMT_LPTSTR, error_code, message);
    if( buffer)
      LocalFree( buffer);
    return 0;
  }
#else // MI_PLATFORM_WINDOWS
  void* handle = dlopen( filename, RTLD_LAZY);
  if( !handle) {
    fprintf( stderr, "%s\n", dlerror());
    return 0;
  }
  void* symbol = dlsym( handle, "mi_factory");
  if( !symbol) {
    fprintf( stderr, "%s\n", dlerror());
    return 0;
  }
#endif // MI_PLATFORM_WINDOWS
  g_dso_handle = handle;

 mi::neuraylib::INeuray* neuray = mi::neuraylib::mi_factory<mi::neuraylib::INeuray>( symbol);
  if( !neuray)
  {
    mi::base::Handle<mi::neuraylib::IVersion> version(
```

```
      mi::neuraylib::mi_factory<mi::neuraylib::IVersion>( symbol));
    if( !version)
      fprintf( stderr, "Error: Incompatible library.\n");
    else
     fprintf( stderr, "Error: Library version %s does not match header version %s.\n",
      version->get_product_version(), MI_NEURAYLIB_PRODUCT_VERSION_STRING);
    return 0;
  }

  check_success( authenticate( neuray) == 0);
  return neuray;
}

bool unload()
{
#ifdef MI_PLATFORM_WINDOWS
  int result = FreeLibrary( (HMODULE)g_dso_handle);
  if( result == 0) {
    LPTSTR buffer = 0;
    LPTSTR message = TEXT("unknown failure");
    DWORD error_code = GetLastError();
    if( FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
      FORMAT_MESSAGE_IGNORE_INSERTS, 0, error_code,
      MAKELANGID( LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR) &buffer, 0, 0))
      message = buffer;
   fprintf( stderr, "Failed to unload library (%u): " FMT_LPTSTR, error_code, message);
    if( buffer)
      LocalFree( buffer);
    return false;
  }
  return true;
#else
  int result = dlclose( g_dso_handle);
  if( result != 0) {
    fprintf( stderr, "%s\n", dlerror());
    return false;
  }
  return true;
#endif
}

void sleep_seconds( mi::Float32 seconds)
{
#ifdef MI_PLATFORM_WINDOWS
  Sleep( static_cast<DWORD>( seconds * 1000));
#else
  usleep( static_cast<useconds_t>( seconds * 1000000));
#endif
}

#ifdef MI_PLATFORM_WINDOWS
#define snprintf _snprintf
#endif

#endif // MI_EXAMPLE_SHARED_H
```

20.24 example_start_shutdown.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"

int main( int /*argc */, char* /*argv*/[])
{
  // Get the INeuray interface in a suitable smart pointer.
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  if( !neuray.is_valid_interface()) {
    fprintf( stderr, "Error: The neuray library failed to load and to provide "
        "the mi::neuraylib::INeuray interface.\n");
    keep_console_open();
    return EXIT_FAILURE;
  }

  // Print library version information.
  mi::base::Handle<const mi::neuraylib::IVersion> version(
    neuray->get_api_component<const mi::neuraylib::IVersion>());

  fprintf( stderr, "MDL SDK header version          = %s\n",
    MI_NEURAYLIB_PRODUCT_VERSION_STRING);
 fprintf( stderr, "MDL SDK library product name   = %s\n", version->get_product_name());
 fprintf( stderr, "MDL SDK library product version = %s\n", version->get_product_version());
 fprintf( stderr, "MDL SDK library build number   = %s\n", version->get_build_number());
 fprintf( stderr, "MDL SDK library build date     = %s\n", version->get_build_date());
 fprintf( stderr, "MDL SDK library build platform  = %s\n", version->get_build_platform());
 fprintf( stderr, "MDL SDK library version string  = \"%s\"\n", version->get_string());

  mi::base::Uuid neuray_id_libraray = version->get_neuray_iid();
  mi::base::Uuid neuray_id_interface = mi::neuraylib::INeuray::IID();

  fprintf( stderr, "MDL SDK header interface ID     = <%2x, %2x, %2x, %2x>\n",
    neuray_id_interface.m_id1,
    neuray_id_interface.m_id2,
    neuray_id_interface.m_id3,
    neuray_id_interface.m_id4);
  fprintf( stderr, "MDL SDK library interface ID    = <%2x, %2x, %2x, %2x>\n\n",
    neuray_id_libraray.m_id1,
    neuray_id_libraray.m_id2,
    neuray_id_libraray.m_id3,
    neuray_id_libraray.m_id4);

  version = 0;

  // configuration settings go here, none in this example

 // After all configurations, neuray is started. A return code of 0 implies success. The start
 // can be blocking or non-blocking. Here the blocking mode is used so that you know that neuray
 // is up and running after the function call. You can use a non-blocking call to do other tasks
  // in parallel and check with
```

```
  //
  //      neuray->get_status() == mi::neuraylib::INeuray::STARTED
  //
  // if startup is completed.
  mi::Sint32 result = neuray->start( true);
  check_start_success( result);

  // scene graph manipulations and rendering calls go here, none in this example.

  // Shutting down in blocking mode. Again, a return code of 0 indicates success.
  check_success( neuray->shutdown( true) == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.25 example_subdivision_surface.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>

mi::neuraylib::ISubdivision_surface* create_cube( mi::neuraylib::ITransaction* transaction)
{
  const mi::Size n_points = 8;
  const mi::Size n_quads = 6;

  mi::Float32_3 cube_points[n_points] = {
    mi::Float32_3( -0.5, -0.5, -0.5),
    mi::Float32_3(  0.5, -0.5, -0.5),
    mi::Float32_3(  0.5,  0.5, -0.5),
    mi::Float32_3( -0.5,  0.5, -0.5),
    mi::Float32_3( -0.5, -0.5,  0.5),
    mi::Float32_3(  0.5, -0.5,  0.5),
    mi::Float32_3(  0.5,  0.5,  0.5),
    mi::Float32_3( -0.5,  0.5,  0.5) };

  mi::Uint32 cube_quads[n_quads][4] = {
    { 0, 1, 5, 4 },
    { 4, 5, 6, 7 },
    { 7, 6, 2, 3 },
    { 3, 2, 1, 0 },
    { 1, 2, 6, 5 },
    { 3, 0, 4, 7 } };

  // Create an empty subdivision surface
  mi::neuraylib::ISubdivision_surface* mesh
   = transaction->create<mi::neuraylib::ISubdivision_surface>( "Subdivision_surface");

  // Create a cube (points and polygons)
  mesh->reserve_points( n_points);
  for( mi::Uint32 i = 0; i < n_points; ++i)
    mesh->append_point( cube_points[i]);
  for( mi::Uint32 i = 0; i < n_quads; ++i)
    mesh->add_polygon( 4);

  // Map vertices of the polygons to points
  mi::base::Handle<mi::neuraylib::IPolygon_connectivity> mesh_connectivity(
    mesh->edit_mesh_connectivity());
  for( mi::Uint32 i = 0; i < n_quads; ++i)
   mesh_connectivity->set_polygon_indices( mi::neuraylib::Polygon_handle( i), cube_quads[i]);
  check_success( mesh->attach_mesh_connectivity( mesh_connectivity.get()) == 0);

  // Set crease values to 1.0f on two opposing faces to end up with a cylinder
  mi::Float32 crease_values[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

```
  mesh->set_crease_values( mi::neuraylib::Polygon_handle( 1), crease_values);
  mesh->set_crease_values( mi::neuraylib::Polygon_handle( 3), crease_values);

  return mesh;
}

void setup_scene( mi::neuraylib::ITransaction* transaction, const char* rootgroup)
{
 // Remove the existing cube from the scene. The unrefined control mesh (in red) will take its
  // place.
  mi::base::Handle<mi::neuraylib::IGroup> group(
    transaction->edit<mi::neuraylib::IGroup>( rootgroup));
  group->detach( "cube_instance");
  group = 0;

  // Create the subdivision surface
 mi::base::Handle<mi::neuraylib::ISubdivision_surface> mesh( create_cube( transaction));
  transaction->store( mesh.get(), "mesh");

 // Create three instances of the subdivision surface with different materials, approximation
  // levels and world-to-object transformation matrices.
  const char* names[3]               = { "instance0", "instance1", "instance2" };
 const char* materials[3]       = { "red_material", "green_material", "blue_material" };
  mi::Float32 approximation_level[3] = { 0.0f, 1.0f, 2.0f };
  const mi::Float32_3 translation[3] = {
    mi::Float32_3(  1.1f, -0.5f,  0.9f),
    mi::Float32_3( -0.9f, -0.5f,  0.9f),
    mi::Float32_3( -0.9f, -0.5f, -1.1f)
  };

  for( mi::Size i = 0; i < 3; ++i) {

    mi::base::Handle<mi::neuraylib::IInstance> instance(
      transaction->create<mi::neuraylib::IInstance>( "Instance"));
    instance->attach( "mesh");

    mi::Float64_4_4 matrix( 1.0);
    matrix.translate( translation[i]);
    matrix.rotate( 0.0, 0.25 * MI_PI_2, 0.0);
    instance->set_matrix( matrix);

    mi::base::Handle<mi::IBoolean> visible(
      instance->create_attribute<mi::IBoolean>( "visible", "Boolean"));
    visible->set_value( true);

    mi::base::Handle<mi::IRef> material(
      instance->create_attribute<mi::IRef>( "material", "Ref"));
    material->set_reference( materials[i]);

    mi::base::Handle<mi::IStructure> approx(
      instance->create_attribute<mi::IStructure>( "approx", "Approx"));
    mi::base::Handle<mi::ISint8> method( approx->get_value<mi::ISint8>( "method"));
    method->set_value( 0);
   mi::base::Handle<mi::IFloat32> const_u( approx->get_value<mi::IFloat32>( "const_u"));
    const_u->set_value( approximation_level[i]);

    transaction->store( instance.get(), names[i]);
```

```
    mi::base::Handle<mi::neuraylib::IGroup> group(
      transaction->edit<mi::neuraylib::IGroup>( rootgroup));
    group->attach( names[i]);
  }
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Add three instances of a subdivision surface with different approximation levels
  setup_scene( transaction.get(), import_result->get_rootgroup());

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
```

```
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_subdivision_surface.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_subdivision_surface <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
```

```
  return EXIT_SUCCESS;
}
```

20.26 example_triangle_mesh.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>
#include <map>
#include <vector>

mi::neuraylib::ITriangle_mesh* create_tetrahedron( mi::neuraylib::ITransaction* transaction)
{
  // Some constants for the vertices, normals, and faces of the tetrahedron
  mi::Float32_3 tetra_points[4] = {
    mi::Float32_3( -0.5, -0.5, -0.5),
    mi::Float32_3(  0.5, -0.5, -0.5),
    mi::Float32_3( -0.5,  0.5, -0.5),
    mi::Float32_3( -0.5, -0.5,  0.5) };

  mi::Float32_3 tetra_normals[4] = {
    mi::Float32_3( -0.577f, -0.577f, -0.577f),
    mi::Float32_3(  0.89f,  -0.20f,  -0.20f),
    mi::Float32_3( -0.20f,   0.89f,  -0.20f),
    mi::Float32_3( -0.20f,  -0.20f,   0.89f) };

  mi::neuraylib::Triangle_point_indices tetra_triangles[4] = {
    mi::neuraylib::Triangle_point_indices( 0, 2, 1),
    mi::neuraylib::Triangle_point_indices( 0, 1, 3),
    mi::neuraylib::Triangle_point_indices( 0, 3, 2),
    mi::neuraylib::Triangle_point_indices( 1, 2, 3) };

  // Create an empty triangle mesh
  mi::neuraylib::ITriangle_mesh* mesh
    = transaction->create<mi::neuraylib::ITriangle_mesh>( "Triangle_mesh");
  check_success( mesh);

  // Create a tetrahedron
  mesh->reserve_points( 4);
  for( mi::Uint32 i = 0; i < 4; ++i)
    mesh->append_point( tetra_points[i]);
  mesh->reserve_triangles( 4);
  for( mi::Uint32 i = 0; i < 4; ++i)
    mesh->append_triangle( tetra_triangles[i]);

  // Use the mesh connectivity for normal vectors
  mi::base::Handle<mi::neuraylib::ITriangle_connectivity> mesh_connectivity(
    mesh->edit_mesh_connectivity());

  // Create an attribute vector for the normals
  mi::base::Handle<mi::neuraylib::IAttribute_vector> normals(
    mesh_connectivity->create_attribute_vector( mi::neuraylib::ATTR_NORMAL));
```

```
  for( mi::Uint32 i = 0; i < 4; ++i)
    normals->append_vector3( tetra_normals[i]);
  check_success( normals->is_valid_attribute());
  check_success( mesh_connectivity->attach_attribute_vector( normals.get()) == 0);
  check_success( !normals->is_valid_attribute());

  check_success( mesh->attach_mesh_connectivity( mesh_connectivity.get()) == 0);

  return mesh;
}

struct Edge {
  mi::Uint32 v1; // smaller index of the two vertex indices
  mi::Uint32 v2; // larger index of the two vertex indices
  Edge() : v1( 0), v2( 0) {}
  Edge( mi::Uint32 p, mi::Uint32 q) : v1( p<q ? p : q), v2( p<q ? q : p) {}
 bool operator< ( const Edge& e) const { return v1 < e.v1 || ( v1 == e.v1 && v2 < e.v2); }
};

void loop_subdivision( mi::neuraylib::ITriangle_mesh* mesh)
{
 // Keep the old mesh sizes in local variables. The old mesh will remain in its place as long as
  // needed, while new elements are appended or kept in temporary arrays.
  mi::Uint32 n = mesh->points_size();    // # points
  mi::Uint32 t = mesh->triangles_size(); // # triangles
  mi::Uint32 e = t * 3 / 2;              // # edges
  mesh->reserve_points( n + e);
  mesh->reserve_triangles( 4 * t);

  // Temporary space for smoothed points for the old existing vertices.
  std::vector< mi::Float32_3 > smoothed_point(
    n, mi::Float32_3( 0.0, 0.0, 0.0));

  // Valence (i.e., vertex degree) of the old existing vertices.
  std::vector< mi::Uint32> valence( n, 0);

 // Edge bisection introduces a single new point per edge, but we will in the course of the
 // algorithm see the edge twice, once per incident triangle. We store a mapping of edges to new
  // vertex indices for simplicity in the following STL map.
  std::map< Edge, mi::Uint32> split_vertex;

  // Compute, with a loop over all old triangles:
  //   - valence of the old vertices
  //   - contribution of 1-ring neighborhood to smoothed old vertices
  //     (weighting by valence follows later)
  //   - new vertices on split edges
  //   - 1:4 split, each triangle is split into 4 triangles
  for( mi::Uint32 i = 0; i < t; ++i) {
    mi::neuraylib::Triangle_point_indices triangle
      = mesh->triangle_point_indices( mi::neuraylib::Triangle_handle( i));

    // Increment valence for each vertex
    ++ valence[ triangle[0]];
    ++ valence[ triangle[1]];
    ++ valence[ triangle[2]];

   // Add neighbor vertices to smoothed vertex following triangle orientation. The opposite
```

```
    // contribution follows from the adjacent triangle.
    mi::Float32_3 p;
    mesh->point( triangle[0], p);
    smoothed_point[ triangle[1]] += p;
    mesh->point( triangle[1], p);
    smoothed_point[ triangle[2]] += p;
    mesh->point( triangle[2], p);
    smoothed_point[ triangle[0]] += p;

    // Determine new vertices at split edges. Loop over all three edges.
    mi::Uint32 new_index[3]; // indices of the three new vertices
    for( mi::Uint32 j = 0; j != 3; ++j) {
      // Consider the edge from v1 to v2.
      mi::Uint32 v0 = triangle[ j     ]; // vertex opposite of edge
      mi::Uint32 v1 = triangle[(j+1)%3]; // vertex that starts the edge
      mi::Uint32 v2 = triangle[(j+2)%3]; // vertex that ends the edge
      Edge edge( v1, v2);
    // Create the new point (or the second half of the contribution) for the split vertex.
      mi::Float32_3 p0, p1;
      mesh->point( v0, p0); // point opposite of edge
      mesh->point( v1, p1); // point that starts the edge
      mi::Float32_3 new_point = ( p0 + p1 * 3.0) / 8.0;
      // Is the split vertex on the edge defined?
    std::map< Edge, mi::Uint32>::iterator split_vertex_pos = split_vertex.find( edge);
      if ( split_vertex_pos == split_vertex.end()) {
        // If not yet defined, create it and a corresponding new vertex in the mesh.
        new_index[j] = mesh->append_point( new_point);
        split_vertex[ edge] = new_index[j];
      } else {
        // If is defined, add the second half of the new vertex contribution
        new_index[j] = split_vertex_pos->second;
        mi::Float32_3 q;
        mesh->point( new_index[j], q);
        mesh->set_point( new_index[j], q + new_point);
      }
    }

    // 1:4 split, each triangle is split into 4 triangles
    mesh->append_triangle(
    mi::neuraylib::Triangle_point_indices( triangle[0], new_index[2], new_index[1]));
    mesh->append_triangle(
    mi::neuraylib::Triangle_point_indices( triangle[1], new_index[0], new_index[2]));
    mesh->append_triangle(
    mi::neuraylib::Triangle_point_indices( triangle[2], new_index[1], new_index[0]));
    mesh->set_triangle( mi::neuraylib::Triangle_handle( i),
    mi::neuraylib::Triangle_point_indices( new_index[0], new_index[1], new_index[2]));
  }

  // One loop over all old vertices combines the 1-ring neighborhood of the old vertices stored in
  // the smoothed vertices, weighted by valence, with the old vertices.
  for( mi::Uint32 i = 0; i < n; ++i) {
    mi::Float32_3 p;
    mesh->point( i, p);
    // Weight used to smooth the old vertices.
    // (An improved implementation would store the weights in a lookup table.)
    mi::Float64 w = 3.0/8.0 + 1.0/4.0 * cos( 2.0 * MI_PI / valence[i]);
    w = 5.0/8.0 - w * w; // final weight: w for 1-ring, 1-w for old vertex
```

```
    mesh->set_point( i,
      (1 - w) * p + w * smoothed_point[i] / static_cast<mi::Float32>( valence[i]));
  }

  // Recompute the normals. They are stored per-point in this example, hence, retrieve them from
  // the mesh connectivity.
  mi::base::Handle<mi::neuraylib::ITriangle_connectivity> mesh_connectivity(
    mesh->edit_mesh_connectivity());
  mi::base::Handle<mi::neuraylib::IAttribute_vector> normals(
    mesh_connectivity->edit_attribute_vector( mi::neuraylib::ATTR_NORMAL));
  check_success( normals.is_valid_interface());
  normals->reserve( n + e);

  // Compute smoothed normal vectors per vertex by averaging adjacent facet normals.
  // First reset all old normals and add space for new normals.
  mi::Uint32 new_n = mesh->points_size();    // # new points
  for( mi::Uint32 i = 0; i < n; ++i)
    normals->set_vector3( i, mi::Float32_3( 0.0, 0.0, 0.0));
  for( mi::Uint32 i = n; i < new_n; ++i)
    normals->append_vector3( mi::Float32_3( 0.0, 0.0, 0.0));

  // Compute, with a loop over all old and all new triangles the normal vectors for each triangle
  // and add them to the per-vertex normals.
  mi::Uint32 new_t = mesh->triangles_size(); // # new triangles
  for( mi::Uint32 i = 0; i < new_t; ++i) {
    mi::neuraylib::Triangle_point_indices triangle
    = mesh_connectivity->triangle_point_indices( mi::neuraylib::Triangle_handle( i));
    mi::Float32_3 p0, p1, p2;
    mesh->point( triangle[0], p0);
    mesh->point( triangle[1], p1);
    mesh->point( triangle[2], p2);
    mi::Float32_3 v = cross( p1 - p0, p2 - p0);
    v.normalize();
    normals->set_vector3( triangle[0],
      v + mi::Float32_3( normals->get_vector3( triangle[0])));
    normals->set_vector3( triangle[1],
      v + mi::Float32_3( normals->get_vector3( triangle[1])));
    normals->set_vector3( triangle[2],
      v + mi::Float32_3( normals->get_vector3( triangle[2])));
  }
  // Renormalize all normals
  for( mi::Uint32 i = 0; i < new_n; ++i) {
    mi::Float32_3 v = normals->get_vector3( i);
    v.normalize();
    normals->set_vector3( i, v);
  }

  // Reattach the normal vector and the mesh connectivity
  mesh_connectivity->attach_attribute_vector( normals.get());
  mesh->attach_mesh_connectivity( mesh_connectivity.get());
}

void setup_scene( mi::neuraylib::ITransaction* transaction, const char* rootgroup)
{
  // Create the red tetrahedron
  mi::base::Handle<mi::neuraylib::ITriangle_mesh> mesh_red( create_tetrahedron( transaction));
  transaction->store( mesh_red.get(), "mesh_red");
```

```
  // Create the instance for the red tetrahedron
  mi::base::Handle<mi::neuraylib::IInstance> instance(
    transaction->create<mi::neuraylib::IInstance>( "Instance"));
  instance->attach( "mesh_red");

  // Set the transformation matrix, the visible attribute, and the material
  mi::Float64_4_4 matrix( 1.0);
  matrix.translate( -0.1, -0.5, 0.2);
  matrix.rotate( 0.0, MI_PI_2, 0.0);
  instance->set_matrix( matrix);

  mi::base::Handle<mi::IBoolean> visible(
    instance->create_attribute<mi::IBoolean>( "visible", "Boolean"));
  visible->set_value( true);

 mi::base::Handle<mi::IRef> material( instance->create_attribute<mi::IRef>( "material", "Ref"));
  material->set_reference( "red_material");

  transaction->store( instance.get(), "instance_red");

  // And attach the instance to the root group
  mi::base::Handle<mi::neuraylib::IGroup> group(
    transaction->edit<mi::neuraylib::IGroup>( rootgroup));
  group->attach( "instance_red");

  // Create the blue object as a Loop-subdivision surface based on the red tetrahedron
  transaction->copy( "mesh_red", "mesh_blue");
  mi::base::Handle<mi::neuraylib::ITriangle_mesh> mesh_blue(
    transaction->edit<mi::neuraylib::ITriangle_mesh>( "mesh_blue"));
  loop_subdivision( mesh_blue.get());
  loop_subdivision( mesh_blue.get());
  loop_subdivision( mesh_blue.get());
  loop_subdivision( mesh_blue.get());

  // Create the instance for the blue object
  instance = transaction->create<mi::neuraylib::IInstance>( "Instance");
  instance->attach( "mesh_blue");

  // Set the transformation matrix, the visible attribute, and the material
  matrix = mi::Float64_4_4( 1.0);
  matrix.translate( 0.4, -1.5, -1.6);
  matrix.rotate( 0.0, 1.25 * MI_PI_2, 0.0);
 mi::Float64_4_4 matrix_scale( 0.25, 0, 0, 0, 0, 0.25, 0, 0, 0, 0, 0.25, 0, 0, 0, 0, 1);
  matrix *= matrix_scale;
  instance->set_matrix( matrix);

  visible = instance->create_attribute<mi::IBoolean>( "visible", "Boolean");
  visible->set_value( true);

  material = instance->create_attribute<mi::IRef>( "material", "Ref");
  material->set_reference( "blue_material");

  transaction->store( instance.get(), "instance_blue");

  // And attach the instance to the root group
  group = transaction->edit<mi::neuraylib::IGroup>( rootgroup);
```

```
  group->attach( "instance_blue");
}

void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Add two triangle meshes to the scene
  setup_scene( transaction.get(), import_result->get_rootgroup());

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
```

```
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_triangle_mesh.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 2) {
    std::cerr << "Usage: example_triangle_mesh <mdl_path>" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.27 example_user_defined_classes.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "imy_class.h"
#include "my_class.h"

void configuration( mi::neuraylib::INeuray* neuray)
{
  // Register the user-defined class.
  mi::base::Handle<mi::neuraylib::IExtension_api> extension_api(
    neuray->get_api_component<mi::neuraylib::IExtension_api>());
  check_success( extension_api.is_valid_interface());
  check_success( extension_api->register_class<My_class>( "My_class") == 0);
}

void test_plugin( mi::neuraylib::INeuray* neuray)
{
 // Get the database, the global scope of the database, and create a transaction in the global
  // scope.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Create instance of My_class and call a method on it.
 mi::base::Handle<IMy_class> my_class( transaction->create<IMy_class>( "My_class"));
  check_success( my_class.is_valid_interface());
  my_class->set_foo( 42);

  // Store instance of My_class in the database and release the handle
  check_success( transaction->store( my_class.get(), "some_name") == 0);
  my_class = 0;

  // Get the instance of My_class from the database again and change the property
  my_class = transaction->edit<IMy_class>( "some_name");
  check_success( my_class.is_valid_interface());
  check_success( my_class->get_foo() == 42);
  my_class->set_foo( 43);
  my_class = 0;

  transaction->commit();
}

int main( int /*argc*/, char* /*argv*/[])
{
  // Access the neuray library
```

```
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get());

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Load and interact with the plugin
  test_plugin( neuray.get());

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.28 example_volumes.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/


#include <mi/neuraylib.h>

#include "example_shared.h"
#include "example_render_target_simple.h"

#include <iostream>

void setup_scene( mi::neuraylib::INeuray* neuray,
         mi::neuraylib::ITransaction* transaction,
         const char* rootgroup,
         const char* volume_file)
{
  // The "unique" element names used in the example
  const char* element_name = "Smoke";
  const char* instance_name = "Smoke_instance";
  const char* vol_name = "Vol";
  const char* tex_name = "Vol_tex";

  // Create volume data container
  mi::base::Handle<mi::neuraylib::IVolume_data> volume_data (
    transaction->create<mi::neuraylib::IVolume_data>("Volume_data") );

  // List VDB content
 mi::base::Handle<const mi::IArray> channel_info(volume_data->list_contents( volume_file));
  fprintf(stderr, "-------------------------------------------------------------\n");
  fprintf(stderr, "Volume file '%s' grid info\n", volume_file);
  fprintf(stderr, "-------------------------------------------------------------\n");
  for (mi::Size i = 0; i < channel_info->get_length(); ++i) {
    mi::base::Handle<const mi::neuraylib::IVolume_info> item(
      channel_info->get_element<mi::neuraylib::IVolume_info>( i));
    const mi::Voxel_block_struct& vb = item->get_data_bounds();
    const mi::Float32_4_4& m = item->get_transform();
    fprintf(stderr,
      "name='%s', value type='%s', data bounds=((%d, %d, %d), (%d, %d, %d)), "
     "transform=(%f, %f, %f, %f,  %f, %f, %f, %f,  %f, %f, %f, %f,  %f, %f, %f, %f)\n",
      item->get_name() ? item->get_name() : "",
      item->get_value_type(),
      vb.min.x, vb.min.y, vb.min.z, vb.max.x, vb.max.y, vb.max.z,
    m.xx, m.xy, m.xz, m.xw, m.yx, m.yy, m.yz, m.yw, m.zx, m.zy, m.zz, m.zw, m.wx, m.wy, m.wz, m.ww);
  }
  fprintf(stderr, "-------------------------------------------------------------\n");

  // Load the volume data
  // nullptr in second argument -> default channel
  check_success( volume_data->reset_file( volume_file) == 0);
  mi::Voxel_block voxel_bounds = volume_data->get_data_bounds();
  mi::Float32_4_4 voxel_transform = volume_data->get_transform();
 transaction->store(volume_data.get(), vol_name, mi::neuraylib::ITransaction::LOCAL_SCOPE);

  // Create the volume object
```

```
  mi::base::Handle<mi::neuraylib::IVolume> volume (
    transaction->create<mi::neuraylib::IVolume>("Volume") );
  mi::Bbox3 bbox ( voxel_bounds );
  volume->set_bounds( bbox, voxel_transform);
  transaction->store( volume.get(), element_name);

  // Create an instance of that volume
  mi::base::Handle<mi::neuraylib::IInstance> instance (
    transaction->create<mi::neuraylib::IInstance>("Instance") );
  instance->attach( element_name);

  mi::base::Handle<mi::IBoolean> visible(
    instance->create_attribute<mi::IBoolean>( "visible", "Boolean"));
  visible->set_value( true);

  // Set an instrance transformation to fit the smoke volume to our example scene
  mi::Float64_4_4 matrix ( voxel_transform );
  // Start by inverting the data given transform.
  matrix.invert();
  // Center at origin
  matrix.translate( -voxel_bounds.center().x,
           -voxel_bounds.center().y,
           -voxel_bounds.center().z);
  // Scale relative to the data size, but keep aspect ratio
  mi::Float64_4_4 scale( 3.0f / static_cast<float>(voxel_bounds.extent().y));
  scale.ww = 1.0f; matrix *= scale;
  // Move above ground plane and towards focus
  matrix.translate(-1.0f, 1.5f, 1.0f);
  matrix.invert();
  instance->set_matrix( matrix);

  mi::base::Handle<mi::IRef> material(
    instance->create_attribute<mi::IRef>( "material", "Ref"));
  material->set_reference( "smoke");

  mi::base::Handle<mi::neuraylib::ITexture> texture (
    transaction->create<mi::neuraylib::ITexture>("Texture") );
  texture->set_volume( vol_name);
 transaction->store( texture.get(), tex_name, mi::neuraylib::ITransaction::LOCAL_SCOPE);

  // Attach the texture to the material property "density"
  mi::base::Handle<mi::neuraylib::IMdl_factory> mdl_factory(
    neuray->get_api_component<mi::neuraylib::IMdl_factory>());
  mi::base::Handle<mi::neuraylib::IExpression_factory> expression_factory(
    mdl_factory->create_expression_factory( transaction));
  mi::neuraylib::Argument_editor smoke_material(
    transaction, "smoke", mdl_factory.get());
  check_success( smoke_material.set_value( "density", tex_name) == 0);

 transaction->store( instance.get(), instance_name, mi::neuraylib::ITransaction::LOCAL_SCOPE);

  // Attach the instance to the root group
  mi::base::Handle<mi::neuraylib::IGroup> group(
    transaction->edit<mi::neuraylib::IGroup>( rootgroup));
  group->attach( instance_name);
}
```

```
void configuration( mi::neuraylib::INeuray* neuray, const char* mdl_path)
{
  // Configure the neuray library. Here we set the search path for .mdl files.
  mi::base::Handle<mi::neuraylib::IRendering_configuration> rc(
    neuray->get_api_component<mi::neuraylib::IRendering_configuration>());
  check_success( rc->add_mdl_path( mdl_path) == 0);
  check_success( rc->add_mdl_path( ".") == 0);
  check_success( rc->add_resource_path( ".") == 0);

  // Load the OpenImageIO, Iray Photoreal, and .mi importer plugins.
  mi::base::Handle<mi::neuraylib::IPlugin_configuration> pc(
    neuray->get_api_component<mi::neuraylib::IPlugin_configuration>());
 check_success( pc->load_plugin_library( "nv_openimageio" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "libiray" MI_BASE_DLL_FILE_EXT) == 0);
  check_success( pc->load_plugin_library( "mi_importer" MI_BASE_DLL_FILE_EXT) == 0);
}

void rendering( mi::neuraylib::INeuray* neuray, const char* volume_file)
{
  // Get the database, the global scope of the database, and create a transaction
  // in the global scope for importing the scene file and storing the scene.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
    neuray->get_api_component<mi::neuraylib::IDatabase>());
  check_success( database.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IScope> scope(
    database->get_global_scope());
  mi::base::Handle<mi::neuraylib::ITransaction> transaction(
    scope->create_transaction());
  check_success( transaction.is_valid_interface());

  // Import the scene file
  mi::base::Handle<mi::neuraylib::IImport_api> import_api(
    neuray->get_api_component<mi::neuraylib::IImport_api>());
  check_success( import_api.is_valid_interface());
  mi::base::Handle<const mi::neuraylib::IImport_result> import_result(
    import_api->import_elements( transaction.get(), "file:main.mi"));
  check_success( import_result->get_error_number() == 0);

  // Add the volume to the scene
 setup_scene( neuray, transaction.get(), import_result->get_rootgroup(), volume_file);

  // Create the scene object
  mi::base::Handle<mi::neuraylib::IScene> scene(
    transaction->create<mi::neuraylib::IScene>( "Scene"));
  scene->set_rootgroup(       import_result->get_rootgroup());
  scene->set_options(         import_result->get_options());
  scene->set_camera_instance( import_result->get_camera_inst());
  transaction->store( scene.get(), "the_scene");

  // Create the render context using the Iray Photoreal render mode
  scene = transaction->edit<mi::neuraylib::IScene>( "the_scene");
  mi::base::Handle<mi::neuraylib::IRender_context> render_context(
    scene->create_render_context( transaction.get(), "iray"));
  check_success( render_context.is_valid_interface());
  mi::base::Handle<mi::IString> scheduler_mode( transaction->create<mi::IString>());
  scheduler_mode->set_c_str( "batch");
  render_context->set_option( "scheduler_mode", scheduler_mode.get());
```

```
  scene = 0;

  // Create the render target and render the scene
  mi::base::Handle<mi::neuraylib::IImage_api> image_api(
    neuray->get_api_component<mi::neuraylib::IImage_api>());
  mi::base::Handle<mi::neuraylib::IRender_target> render_target(
    new Render_target( image_api.get(), "Color", 512, 384));
 check_success( render_context->render( transaction.get(), render_target.get(), 0) >= 0);

  // Write the image to disk
  mi::base::Handle<mi::neuraylib::IExport_api> export_api(
    neuray->get_api_component<mi::neuraylib::IExport_api>());
  check_success( export_api.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas( render_target->get_canvas( 0));
  export_api->export_canvas( "file:example_volumes.png", canvas.get());

  transaction->commit();
}

int main( int argc, char* argv[])
{
  // Collect command line parameters
  if( argc != 3 && argc != 2) {
    std::cerr << "Usage: example_volumes <mdl_path> [<volume_file>]" << std::endl;
    keep_console_open();
    return EXIT_FAILURE;
  }
  const char* mdl_path = argv[1];
  const char* volume_file = argc == 2 ? "smoke.vdb" : argv[2];

  // Access the neuray library
  mi::base::Handle<mi::neuraylib::INeuray> neuray( load_and_get_ineuray());
  check_success( neuray.is_valid_interface());

  // Configure the neuray library
  configuration( neuray.get(), mdl_path);

  // Start the neuray library
  mi::Sint32 result = neuray->start();
  check_start_success( result);

  // Do the actual rendering
  rendering( neuray.get(), volume_file);

  // Shut down the neuray library
  check_success( neuray->shutdown() == 0);
  neuray = 0;

  // Unload the neuray library
  check_success( unload());

  keep_console_open();
  return EXIT_SUCCESS;
}
```

20.29 imy_class.h

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#ifndef IMY_CLASS_H
#define IMY_CLASS_H

#include <mi/neuraylib.h>

class IMy_class : public
 mi::base::Interface_declare<0x7274f2a1,0xb7e1,0x4fc7,0x8c,0xe4,0xef,0x2d,0xd0,0x9b,0x75,0x52,
                mi::neuraylib::IUser_class>
{
public:
  // Sets property foo
  virtual void set_foo( mi::Sint32 foo) = 0;

  // Gets property foo
  virtual mi::Sint32 get_foo() const = 0;
};

#endif // IMY_CLASS_H
```

20.30 my_class.h

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#ifndef MY_CLASS_H
#define MY_CLASS_H

#include <mi/neuraylib.h>
#include "imy_class.h"

class My_class : public
 mi::neuraylib::User_class<0x0650d689,0xef50,0x433f,0xb3,0xae,0xae,0x83,0xa9,0xf2,0xf2,0xa6,
              IMy_class>
{
public:
  // Constructor
  My_class() { m_foo = 0; }

  // Destructor
  ~My_class() { }

  // Sets property foo
  void set_foo( mi::Sint32 foo) { m_foo = foo; }

  // Gets property foo
  mi::Sint32 get_foo() const { return m_foo; }

  // Serializes an instance of My_class
  void serialize( mi::neuraylib::ISerializer* serializer) const
  {
    serializer->write( &m_foo);
  }

  // Deserializes an instance of My_class
  void deserialize( mi::neuraylib::IDeserializer* deserializer)
  {
    deserializer->read( &m_foo);
  }

  // Creates a copy of the object
  mi::neuraylib::IUser_class* copy () const
  {
    My_class* copy = new My_class( *this);
    return copy;
  }

  // Returns a human readable class name
  const char* get_class_name() const
  {
    return "My_class";
  }

  // Returns the list of elements referenced by this elements, none in this example.
  mi::IArray* get_references( mi::neuraylib::ITransaction* /*transaction*/) const
  {
```

```
    return 0;
  }

private:
  // Member to realize property foo
  mi::Sint32 m_foo;
};

#endif // MY_CLASS_H
```

20.31 plugin.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/neuraylib.h>
#include "my_class.h"

class Plugin_impl : public mi::neuraylib::IPlugin
{
public:
  // Virtual destructor
  virtual ~Plugin_impl() { }

  // Returns the name of the plugin
  const char* get_name() const { return "example plugin"; }

  // Returns the type of the plugin
  const char* get_type() const { return MI_NEURAYLIB_PLUGIN_TYPE; }

  // Returns the version of the plugin
  mi::Sint32 get_version() const { return 1; }

  // Returns the compiler used to compile the plugin
  const char* get_compiler() const { return "unknown"; }

  // Releases the plugin giving back all allocated resources
  void release() { delete this; }

  // Initializes the plugin
  bool init( mi::neuraylib::IPlugin_api* plugin_api)
  {
    mi::base::Handle<mi::neuraylib::ILogging_configuration> logging_configuration(
      plugin_api->get_api_component<mi::neuraylib::ILogging_configuration>());
    mi::base::Handle<mi::base::ILogger> logger(
      logging_configuration->get_forwarding_logger());
    logger->message( mi::base::MESSAGE_SEVERITY_INFO, "PLUGIN",
      "Plugin_impl::init() called");
    mi::base::Handle<mi::neuraylib::IExtension_api> extension_api(
      plugin_api->get_api_component<mi::neuraylib::IExtension_api>());
    if( ! extension_api.is_valid_interface())
      return false;
    return extension_api->register_class<My_class>( "My_class") == 0;
  }

  // Exits the plugin
  bool exit( mi::neuraylib::IPlugin_api* plugin_api)
  {
    mi::base::Handle<mi::neuraylib::ILogging_configuration> logging_configuration(
      plugin_api->get_api_component<mi::neuraylib::ILogging_configuration>());
    mi::base::Handle<mi::base::ILogger> logger(
      logging_configuration->get_forwarding_logger());
    logger->message( mi::base::MESSAGE_SEVERITY_INFO, "PLUGIN",
      "Plugin_impl::exit() called");
    return true;
  }
```

```
};

extern "C"
MI_DLL_EXPORT
mi::base::Plugin* mi_plugin_factory(
  mi::Sint32 index,         // index of the plugin
  void* context)            // context given to the library, ignore
{
  if( index != 0)
    return 0;
  return new Plugin_impl;
}
```

20.32 vanilla_exporter.h

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/neuraylib.h>

#include <string>
#include <list>
#include <set>
#include <map>
#include <utility>      // for std::pair
#include <sstream>

static std::string enum_to_str( mi::base::Message_severity severity)
{
  switch( severity) {
    case mi::base::MESSAGE_SEVERITY_FATAL:   return "fatal";
    case mi::base::MESSAGE_SEVERITY_ERROR:   return "error";
    case mi::base::MESSAGE_SEVERITY_WARNING: return "warning";
    case mi::base::MESSAGE_SEVERITY_INFO:    return "information";
    case mi::base::MESSAGE_SEVERITY_VERBOSE: return "verbose";
    case mi::base::MESSAGE_SEVERITY_DEBUG:   return "debug";
    default:                                 return "unknown" ;
  }
}

class Vanilla_export_state
 : public mi::base::Interface_implement< mi::neuraylib::IImpexp_state>
{
public:
  // Constructor.
 Vanilla_export_state( const char* uri, const mi::neuraylib::IImpexp_state* parent_state)
   : m_line_number( 1),
    m_uri( uri ? uri : ""),
    m_parent_state( parent_state, mi::base::DUP_INTERFACE) { }

  // Definition of all interface functions.

  const char* get_uri() const { return m_uri.empty() ? 0 : m_uri.c_str(); }

  mi::Uint32  get_line_number() const { return m_line_number; }

  void set_line_number( mi::Uint32 number) { m_line_number = number; }

  void incr_line_number() { ++m_line_number; }

  const mi::neuraylib::IImpexp_state* get_parent_state() const
  {
    if( m_parent_state)
      m_parent_state->retain();
    return m_parent_state.get();
  }

private:
  mi::Uint32 m_line_number;
```

```
  std::string m_uri;
  mi::base::Handle<const mi::neuraylib::IImpexp_state> m_parent_state;
};

class Vanilla_exporter
 : public mi::base::Interface_implement< mi::neuraylib::IExporter>
{
public:
  // Definition of all interface functions.

  mi::neuraylib::IImpexp_state* create_impexp_state(
    const char* uri, const mi::neuraylib::IImpexp_state* parent_state) const
  {
    return new Vanilla_export_state( uri, parent_state);
  }

  // This exporter supports the file name extensions ".vnl" and ".van".
  const char* get_supported_extensions( mi::Uint32 i) const
  {
    switch( i) {
      case 0:  return ".vnl";
      case 1:  return ".van";
      default: return 0;
    }
  }

  mi::neuraylib::Impexp_priority get_priority() const
  {
    return mi::neuraylib::IMPEXP_PRIORITY_WELL_DEFINED;
  }

  const char* get_name() const { return "NVIDIA example vanilla (v1) exporter"; }

  const char* get_author() const { return "NVIDIA Corporation, Berlin, Germany"; }

  mi::base::Uuid get_uuid() const
  {
    mi::base::Uuid uuid;
    uuid.m_id1 = 0x7b0a3b59;
    uuid.m_id2 = 0xbed44329;
    uuid.m_id3 = 0xad1a2353;
    uuid.m_id4 = 0xb0ab89a8;
    return uuid;
  }

  mi::Uint32 get_major_version() const { return 1; }

  mi::Uint32 get_minor_version() const { return 0; }

  bool test_file_type( const char* extension) const;

 bool test_file_type( const char* extension, const mi::neuraylib::IWriter* writer) const;

  mi::neuraylib::IExport_result* export_scene(
    mi::neuraylib::ITransaction* transaction,
    const char* extension,
    mi::neuraylib::IWriter* writer,
```

```
    const char* rootgroup,
    const char* caminst,
    const char* options,
    const mi::IMap* exporter_options,
    mi::neuraylib::IImpexp_state* export_state) const;

  mi::neuraylib::IExport_result* export_elements(
    mi::neuraylib::ITransaction* transaction,
    const char* extension,
    mi::neuraylib::IWriter* writer,
    const mi::IArray* elements,
    const mi::IMap* exporter_options,
    mi::neuraylib::IImpexp_state* export_state) const;

private:
  // Definition of support functions. They are not part of the interface.

  // Formats message with context and appends it to the messages in the result.
  static mi::neuraylib::IExport_result_ext* report_message(
    mi::neuraylib::IExport_result_ext* result,
    mi::Uint32 message_number,
    mi::base::Message_severity message_severity,
    std::string message,
    const mi::neuraylib::IImpexp_state* export_state)
  {
    std::ostringstream s;
    const char* uri = export_state->get_uri();
    s << (uri ? uri : "(no URI)")
     << ":" << export_state->get_line_number() << ": "
     << "Vanilla exporter message " << message_number << ", "
     << "severity " << enum_to_str( message_severity) << ": "
     << message;
    // Report context of all parent export states from recursive invocations of
    // export_elements() in their own lines with indentation.
    mi::base::Handle<const mi::neuraylib::IImpexp_state> parent_state(
      export_state->get_parent_state());
    while( parent_state.is_valid_interface()) {
      s << "\n    included from: " << parent_state->get_uri()
       << ":" << parent_state->get_line_number();
      parent_state = parent_state->get_parent_state();
    }
    result->message_push_back( message_number, message_severity, s.str().c_str());
    return result;
  }

  // Writes name to writer, writes the name without the leading prefix if it is equal to the
  // prefix parameter.
  static void write_name(
   const std::string& name, const std::string& prefix, mi::neuraylib::IWriter* writer)
  {
    if( prefix.size() > 0 && 0 == name.compare( 0, prefix.size(), prefix))
      writer->writeline( name.c_str() + prefix.size());
    else
      writer->writeline( name.c_str());
  }

  // Returns the element type of an element.
```

```
  static std::string get_element_type( const mi::base::IInterface* interface)
  {
    mi::base::Handle<const mi::neuraylib::IGroup> group(
      interface->get_interface<mi::neuraylib::IGroup>());
    if( group.is_valid_interface())
      return "Group";
    mi::base::Handle<const mi::neuraylib::IInstance> instance(
      interface->get_interface<mi::neuraylib::IInstance>());
    if( instance.is_valid_interface())
      return "Instance";
    return "Unknown";
  }
};

bool Vanilla_exporter::test_file_type( const char* extension) const
{
  // This exporter supports the file name extensions ".vnl" and ".van".
  mi::Size len = std::strlen( extension);
  return (len > 3)
    &&  (( 0 == strcmp( extension + len - 4, ".vnl"))
     || ( 0 == strcmp( extension + len - 4, ".van")));
}

bool Vanilla_exporter::test_file_type(
  const char* extension, const mi::neuraylib::IWriter*) const
{
  // The writer capabilities do not matter for this simple format. More involved formats might
  // require random access from the writer.
  return test_file_type( extension);
}

mi::neuraylib::IExport_result* Vanilla_exporter::export_scene(
  mi::neuraylib::ITransaction* transaction,
  const char* /*extension*/,
  mi::neuraylib::IWriter* writer,
  const char* rootgroup,
  const char* caminst,
  const char* options,
  const mi::IMap* exporter_options,
  mi::neuraylib::IImpexp_state* export_state) const
{
  // Create the exporter result instance for the return value. If that fails something is really
  // wrong and we return 0.
  mi::neuraylib::IExport_result_ext* result
    = transaction->create<mi::neuraylib::IExport_result_ext>( "Export_result_ext");
  if( !result)
    return 0;

  // Get the 'strip_prefix' option.
  std::string strip_prefix;
  if( exporter_options && exporter_options->has_key( "strip_prefix")) {
    mi::base::Handle<const mi::IString> option(
      exporter_options->get_value<mi::IString>( "strip_prefix"));
    if( !option.is_valid_interface())
      return report_message(
        result, 6, mi::base::MESSAGE_SEVERITY_ERROR,
        "The option 'strip_prefix' has an invalid type.", export_state);
```

```
    strip_prefix = option->get_c_str();
  }

  // Two data structures maintain the information during export. The elements list keeps the names
  // of all elements that we want to export and a bit if they have been expanded already. The
  // order of elements follows the .mi requirements; elements that are referenced are first in the
  // list and exported before the elements that reference them. The elements_exported map
  // maintains a list of all exported elements. That allows us to handle objects that are
  // referenced multiple times and export them only once.
  std::list< std::pair< std::string, bool> > elements;
  std::set< std::string>                     elements_exported;

  // Initialize the elements list with the three input parameters.
  if( options && options[0] != '\0')
    elements.push_back( std::make_pair( std::string( options), false));
  if( caminst && caminst[0] != '\0')
    elements.push_back( std::make_pair( std::string( caminst), false));
  elements.push_back( std::make_pair( std::string( rootgroup), false));

  // Start file with magic header and use Windows line-ending convention with CR LF pairs.
  writer->writeline( "VANILLA\r\n");

  // Main loop through all elements
  // This is a simplified recursive directed acyclic graph traversal on the scene graph that
  // performs a depth first search. The traversal is implemented as an iterative loop and a stack
  // on the elements list data structure. The boolean value of entries in the elements list
  // encodes whether we are descending or are ascending in the graph. This flag determines whether
  // we need to expand into the element and put all its children on the stack, or whether we are
  // done with the element and can write it out to file. Other exporters might need to manage more
  // data during the traversal, such as a transformation stack.
  while( (0 == writer->get_error_number()) && (!elements.empty())) {
    if( elements.front().second) {

      // Traversal is ascending in the scene graph
      // Keep element name and remove it from list
      std::string name = elements.front().first;
      elements.pop_front();
      // Check if element has not been written yet
      if( elements_exported.find( name) == elements_exported.end()) {
        // Element can be written to file, mark it as written
        elements_exported.insert( name);
        // Access the element in the DB
        mi::base::Handle<const mi::base::IInterface> element(
          transaction->access( name.c_str()));
        if( !element.is_valid_interface()) {
          // The element is not in the DB. Export fails with customized message.
          std::string message( "Element '");
          message += name + "' does not exist in database, export failed.";
          // Error numbers from 6000 to 7999 are reserved for custom
          // exporter messages like this one
          return report_message( result, 6001, mi::base::MESSAGE_SEVERITY_ERROR,
            message, export_state);
        }
        writer->writeline( get_element_type( element.get()).c_str());
        writer->writeline( " \"");
        write_name( name, strip_prefix, writer);
        writer->writeline( "\"\r\n");
```

```
      }

    } else {

      // Traversal is descending in the scene graph, mark element as expanded.
      elements.front().second = true;
      // Expand front element, but keep it in the list.
      std::string name = elements.front().first;
      // Access the element in the DB.
      mi::base::Handle<const mi::base::IInterface> element(
        transaction->access( name.c_str()));
      if( !element.is_valid_interface()) {
        // The element is not in the DB. Export fails with customized message.
        std::string message( "Element '");
        message += name + "' does not exist in database, export failed.";
        return report_message( result, 6002, mi::base::MESSAGE_SEVERITY_ERROR,
          message, export_state);
      }
      // Dispatch on the type name of the element.
      std::string element_type = get_element_type( element.get());
      if( element_type == "Group") {

        mi::base::Handle<const mi::neuraylib::IGroup> group(
          element->get_interface<mi::neuraylib::IGroup>());
        // Enumerate all elements in the group and push them in reverse order on the
        // elements list front.
        mi::Uint32 group_size = group->get_length();
        for( mi::Uint32 i = 0; i != group_size; ++i) {
          const char* element_name = group->get_element( group_size - i -1);
        // Optimization: put name only in the elements list if it has not been exported
          // yet.
          if( elements_exported.find( element_name) == elements_exported.end())
           elements.push_front( std::make_pair( std::string( element_name), false));
        }

      } else if( element_type == "Instance") {

        mi::base::Handle<const mi::neuraylib::IInstance> instance(
          element->get_interface<mi::neuraylib::IInstance>());
        // Get element in the instance and push it on the elements list front.
        const char* element_name = instance->get_item();
     // Optimization: put name only in the elements list if it has not been exported yet.
        if( elements_exported.find( element_name) == elements_exported.end())
          elements.push_front( std::make_pair( std::string( element_name), false));

      }
    }
  }

  // Report message condition for a possibly failed writer call
  if( writer->get_error_number() != 0)
    return report_message(
      result,
      static_cast<mi::Uint32>( writer->get_error_number()),
      mi::base::MESSAGE_SEVERITY_ERROR,
      writer->get_error_message() ? writer->get_error_message() : "",
      export_state);
```

```
  return result;
}

mi::neuraylib::IExport_result* Vanilla_exporter::export_elements(
  mi::neuraylib::ITransaction* transaction,
  const char* /*extension*/,
  mi::neuraylib::IWriter* writer,
  const mi::IArray* elements,
  const mi::IMap* exporter_options,
  mi::neuraylib::IImpexp_state* export_state) const
{
  // Create the exporter result instance for the return value.
  // If that fails something is really wrong and we return 0.
  mi::neuraylib::IExport_result_ext* result
    = transaction->create<mi::neuraylib::IExport_result_ext>( "Export_result_ext");
  if( !result)
    return 0;

  // Get the 'strip_prefix' option.
  std::string strip_prefix;
  if( exporter_options && exporter_options->has_key( "strip_prefix")) {
    mi::base::Handle<const mi::IString> option(
      exporter_options->get_value<mi::IString>( "strip_prefix"));
    if( !option.is_valid_interface())
      return report_message(
        result, 6, mi::base::MESSAGE_SEVERITY_ERROR,
        "The option 'strip_prefix' has an invalid type.", export_state);
    strip_prefix = option->get_c_str();
  }

 // Start file with magic header and use Windows line-ending convention with CR LF pairs.
  writer->writeline( "VANILLA\x0D\x0A");

  // Iterate through the string array of element names
  mi::Size size = elements->get_length();
  for( mi::Size i = 0; (0 == writer->get_error_number()) && i < size; ++i) {

    // Get string for element i from the array
   mi::base::Handle<const mi::IString> name( elements->get_element<mi::IString>( i));
    if( !name.is_valid_interface()) {
      return report_message( result, 6007, mi::base::MESSAGE_SEVERITY_ERROR,
        "element array contains an invalid object", export_state);
    }
    const char* element_name = name->get_c_str();

    // Access the element in the DB
   mi::base::Handle<const mi::base::IInterface> element( transaction->access( element_name));
    if( !element.is_valid_interface()) {
      std::string message( "Element '");
      message += element_name;
      message += "' does not exist in database, export failed.";
      return report_message( result, 6008, mi::base::MESSAGE_SEVERITY_ERROR,
        message, export_state);
    }

    // Write element to file
```

```
    writer->writeline( get_element_type( element.get()).c_str());
    writer->writeline( " \"");
    write_name( element_name, strip_prefix, writer);
    writer->writeline( "\"\x0D\x0A");
  }

  return result;
}
```

20.33 vanilla_importer.h

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/neuraylib.h>

#include <string>
#include <sstream>

static std::string enum_to_str( mi::base::Message_severity severity)
{
  switch( severity) {
    case mi::base::MESSAGE_SEVERITY_FATAL:   return "fatal";
    case mi::base::MESSAGE_SEVERITY_ERROR:   return "error";
    case mi::base::MESSAGE_SEVERITY_WARNING: return "warning";
    case mi::base::MESSAGE_SEVERITY_INFO:    return "information";
    case mi::base::MESSAGE_SEVERITY_VERBOSE: return "verbose";
    case mi::base::MESSAGE_SEVERITY_DEBUG:   return "debug";
    default:                                 return "unknown" ;
  }
}

class Vanilla_import_state
 : public mi::base::Interface_implement< mi::neuraylib::IImpexp_state>
{
public:
  // Constructor.
 Vanilla_import_state( const char* uri, const mi::neuraylib::IImpexp_state* parent_state)
   : m_line_number( 1),
    m_uri( uri ? uri : ""),
    m_parent_state( parent_state, mi::base::DUP_INTERFACE) { }

  // Definition of all interface functions.

  const char* get_uri() const { return m_uri.empty() ? 0 : m_uri.c_str(); }

  mi::Uint32 get_line_number() const { return m_line_number; }

  void set_line_number( mi::Uint32 number) { m_line_number = number; }

  void incr_line_number() { ++m_line_number; }

  const mi::neuraylib::IImpexp_state* get_parent_state() const
  {
    if( m_parent_state)
      m_parent_state->retain();
    return m_parent_state.get();
  }

private:
  mi::Uint32 m_line_number;
  std::string m_uri;
  mi::base::Handle<const mi::neuraylib::IImpexp_state> m_parent_state;
};
```

```
class Vanilla_importer
 : public mi::base::Interface_implement< mi::neuraylib::IImporter>
{
public:
  // Constructor.
  Vanilla_importer( mi::neuraylib::IPlugin_api* plugin_api)
    : m_plugin_api( plugin_api, mi::base::DUP_INTERFACE) { }

  // Definition of all interface functions.

  mi::neuraylib::IImpexp_state* create_impexp_state(
    const char* uri, const mi::neuraylib::IImpexp_state* parent_state) const
  {
    return new Vanilla_import_state( uri, parent_state);
  }

  // This importer supports the file name extensions ".vnl" and ".van".
  const char* get_supported_extensions( mi::Uint32 i) const
  {
    switch( i) {
      case 0:  return ".vnl";
      case 1:  return ".van";
      default: return 0;
    }
  }

  mi::neuraylib::Impexp_priority get_priority() const
  {
    return mi::neuraylib::IMPEXP_PRIORITY_WELL_DEFINED;
  }

  const char* get_name() const { return "NVIDIA example vanilla (v1) importer"; }

  const char* get_author() const { return "NVIDIA Corporation, Berlin, Germany"; }

  mi::base::Uuid get_uuid() const
  {
    mi::base::Uuid uuid;
    uuid.m_id1 = 0x338eca60;
    uuid.m_id2 = 0x31004802;
    uuid.m_id3 = 0xaab9046b;
    uuid.m_id4 = 0x9e0b1d9b;
    return uuid;
  }

  mi::Uint32 get_major_version() const { return 1; }

  mi::Uint32 get_minor_version() const { return 0; }

  bool test_file_type( const char* extension) const;

 bool test_file_type( const char* extension, const mi::neuraylib::IReader* reader) const;

  mi::neuraylib::IImport_result* import_elements(
    mi::neuraylib::ITransaction* transaction,
    const char* extension,
    mi::neuraylib::IReader* reader,
```

```
    const mi::IMap* options,
    mi::neuraylib::IImpexp_state* import_state) const;

private:
  // Definition of support functions. They are not part of the interface.

  // Format message with context and append it to the messages in the result.
  static mi::neuraylib::IImport_result_ext* report_message(
    mi::neuraylib::IImport_result_ext* result,
    mi::Uint32 message_number,
    mi::base::Message_severity message_severity,
    std::string message,
    const mi::neuraylib::IImpexp_state* import_state)
  {
    std::ostringstream s;
    s << import_state->get_uri()
     << ":" << import_state->get_line_number() << ": "
     << "Vanilla importer message " << message_number << ", "
     << "severity " << enum_to_str( message_severity) << ": "
     << message;
    // Report context of all parent import states from recursive invocations of
    // import_elements() in their own lines with indentation.
    mi::base::Handle<const mi::neuraylib::IImpexp_state> parent_state(
      import_state->get_parent_state());
    while( parent_state.is_valid_interface()) {
      s << "\n    included from: " << parent_state->get_uri()
       << ":" << parent_state->get_line_number();
      parent_state = parent_state->get_parent_state();
    }
    result->message_push_back( message_number, message_severity, s.str().c_str());
    return result;
  }

  mi::base::Handle<mi::neuraylib::IPlugin_api> m_plugin_api;
};

bool Vanilla_importer::test_file_type( const char* extension) const
{
  // This importer supports the file name extensions ".vnl" and ".van".
  mi::Size len = std::strlen( extension);
  return (len > 3)
    &&  (( 0 == strcmp( extension + len - 4, ".vnl"))
     || ( 0 == strcmp( extension + len - 4, ".van")));
}

bool Vanilla_importer::test_file_type(
  const char* extension, const mi::neuraylib::IReader* reader) const
{
  // Use magic header check if lookahead is available
  if( reader->supports_lookahead()) {
   // File has to start with "VANILLA" and linebreak, which can be \n or \r depending on the
    // line ending convention in the file.
    const char** buffer = 0;
    mi::Sint64 n = reader->lookahead( 8, buffer);
    return ( n >= 8) && (0 == std::strncmp( *buffer, "VANILLA", 7))
            && (((*buffer)[7] == '\n') || ((*buffer)[7] == '\r'));
  }
```

```
  // This importer supports the file name extensions ".vnl" and ".van".
  mi::Size len = std::strlen( extension);
  return (len > 3)
    &&  (( 0 == strcmp( extension + len - 4, ".vnl"))
     || ( 0 == strcmp( extension + len - 4, ".van")));
}

mi::neuraylib::IImport_result* Vanilla_importer::import_elements(
  mi::neuraylib::ITransaction* transaction,
  const char* extension,
  mi::neuraylib::IReader* reader,
  const mi::IMap* importer_options,
  mi::neuraylib::IImpexp_state* import_state) const
{
 // Create the importer result instance for the return value. If that fails something is really
  // wrong and we return 0.
  mi::neuraylib::IImport_result_ext* result
    = transaction->create<mi::neuraylib::IImport_result_ext>( "Import_result_ext");
  if( !result)
    return 0;

  // Get the 'prefix' option.
  std::string prefix;
  if( importer_options && importer_options->has_key( "prefix")) {
    mi::base::Handle<const mi::IString> option(
      importer_options->get_value<mi::IString>( "prefix"));
    prefix = option->get_c_str();
  }

  // Get the 'list_elements' option.
  bool list_elements = false;
  if( importer_options && importer_options->has_key( "list_elements")) {
    mi::base::Handle<const mi::IBoolean> option(
      importer_options->get_value<mi::IBoolean>( "list_elements"));
    list_elements = option->get_value<bool>();
  }

 // Before we start parsing the file, we create a group that collects all top-level elements.
  // This will be our rootgroup.
  std::string root_group_name = prefix + "Vanilla::root_group";
  mi::base::Handle<mi::neuraylib::IGroup> rootgroup(
    transaction->create<mi::neuraylib::IGroup>( "Group"));
 mi::Sint32 error_code = transaction->store( rootgroup.get(), root_group_name.c_str());
  if( error_code != 0)
    return report_message( result, 4010, mi::base::MESSAGE_SEVERITY_ERROR,
      "failed to create the root group", import_state);
  rootgroup = transaction->edit<mi::neuraylib::IGroup>( root_group_name.c_str());

  // Register the rootgroup with the importer result.
  result->set_rootgroup( root_group_name.c_str());

 // If the element list flag is set, record the rootgroup element also in the elements array of
  // the result.
  if( list_elements)
    result->element_push_back( root_group_name.c_str());

 // Assume it is a line based text format and read it line by line. Assume lines are no longer
```

```
// than 256 chars, otherwise read it in pieces
char buffer[257];
while( reader->readline( buffer, 257) && buffer[0] != '\0') {

 // Periodically check whether the transaction is still open. If not, stop importing.
  if( !transaction->is_open())
    break;

  // Here you can process the buffer content of size len. It corresponds to the line
  // import_state->get_line_number() of the input file.
  mi::Size len = std::strlen( buffer);

 // We illustrate some actions triggered by fixed line numbers: Line 3 of a ".vnl" file
  // triggers the recursive inclusion of the file "test2.van" file with a prefix.
  mi::Size ext_len = std::strlen( extension);
  if( 3 == import_state->get_line_number()
    && (ext_len > 3)
    && 0 == strcmp( extension + ext_len - 4, ".vnl")) {
    // Get a IImport_api handle to call its import_elements() function
    mi::base::Handle<mi::neuraylib::IImport_api> import_api(
      m_plugin_api->get_api_component<mi::neuraylib::IImport_api>());
    if( !import_api.is_valid_interface())
    // Error numbers from 4000 to 5999 are reserved for custom importer messages like
      // this one
      return report_message( result, 4001, mi::base::MESSAGE_SEVERITY_ERROR,
        "did not get a valid IImport_api object, import failed", import_state);
  // Call the importer recursively to illustrate the handling of include files and similar
   // things. We trigger this import only on a ".vnl" file and include the fixed file
    // "test2.van". We give the included file an extra prefix "Prefix_".
    mi::base::Handle<mi::neuraylib::IFactory> factory(
      m_plugin_api->get_api_component<mi::neuraylib::IFactory>());
 mi::base::Handle<mi::IString> child_prefix( factory->create<mi::IString>( "String"));
    child_prefix->set_c_str( "Prefix_");
    mi::base::Handle<mi::IMap> child_importer_options( factory->clone<mi::IMap>(
      importer_options));
    child_importer_options->erase( "prefix");
    child_importer_options->insert( "prefix", child_prefix.get());
    mi::base::Handle<const mi::neuraylib::IImport_result> include_result(
      import_api->import_elements( transaction, "file:test2.van",
        child_importer_options.get(), import_state));
    // Safety check, if this fails, the import is not continued.
    if( !include_result.is_valid_interface())
      return report_message( result, 4002, mi::base::MESSAGE_SEVERITY_ERROR,
       "import was not able to create result object, import failed", import_state);
   // Process the result. Even in the case of an error, we need to process the elements
    // array.
    if( list_elements)
      result->append_elements( include_result.get());
    // Append messages of include to this result
    result->append_messages( include_result.get());
    // React on errors during processing of the include
    if( include_result->get_error_number() > 0) {
      // Report the failure of the include as a separate message too
      report_message( result, 4003, mi::base::MESSAGE_SEVERITY_ERROR,
        "including file 'test2.van' failed.", import_state);
    } else {
    // Recursive import was successful. The rootgroup of the import is now appended to
```

```
      // this rootgroup
      if( 0 == include_result->get_rootgroup())
        report_message( result, 4004, mi::base::MESSAGE_SEVERITY_ERROR,
          "include file 'test2.van' did not contain a rootgroup", import_state);
      else
        rootgroup->attach( include_result->get_rootgroup());
    }
  }

  // Line 4 triggers several messages and adds an empty group to the rootgroup.
  if( 4 == import_state->get_line_number()) {

    // Several messages, file parsing continues
    report_message( result, 4005, mi::base::MESSAGE_SEVERITY_FATAL,
      "test message in line 4", import_state);
    report_message( result, 4006, mi::base::MESSAGE_SEVERITY_ERROR,
      "test message in line 4", import_state);
    report_message( result, 4007, mi::base::MESSAGE_SEVERITY_WARNING,
      "test message in line 4", import_state);
    report_message( result, 4008, mi::base::MESSAGE_SEVERITY_INFO,
      "test message in line 4", import_state);
    report_message( result, 4009, mi::base::MESSAGE_SEVERITY_VERBOSE,
      "test message in line 4", import_state);

    // Create a group "Vanilla::Group1"
    std::string group_name = prefix + "Vanilla::Group1";
    mi::base::Handle<mi::neuraylib::IGroup> group(
      transaction->create<mi::neuraylib::IGroup>( "Group"));
    mi::Sint32 error_code = transaction->store( group.get(), group_name.c_str());
    if( error_code != 0)
      report_message( result, 4011, mi::base::MESSAGE_SEVERITY_ERROR,
        "unexpected error in line 4", import_state);
    else {
      // Add this group to the rootgroup
      rootgroup->attach( group_name.c_str());
   // If get_list_elements_flag is set, record the new element in the elements array of
      // the result.
      if( list_elements)
        result->element_push_back( group_name.c_str());
    }
  }

  // Handle line numbers, buffer might end in '\n' or not if line was too long.
  if( (len > 0) && ('\n' == buffer[len-1]))
    import_state->incr_line_number();
}

if( reader->eof()) {
  // Normal end
  return result;
}

// Report error condition for a failed reader call
return report_message(
  result,
  static_cast<mi::Uint32>( reader->get_error_number()),
  mi::base::MESSAGE_SEVERITY_ERROR,
```

```
    reader->get_error_message() ? reader->get_error_message() : "",
    import_state);
}
```